Understanding the Evolution of Code Clones in Software Systems

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Avigit Kumar Saha

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Code cloning is a common practice in software development. However, code cloning has both positive aspects such as accelerating the development process and negative aspects such as causing code bloat. After a decade of active research, it is clear that removing all of the clones from a software system is not desirable. Therefore, it is better to manage clones than to remove them. A software system can have thousands of clones in it, which may serve multiple purposes. However, some of the clones may cause unwanted management difficulties and clones like these should be refactored. Failure to manage clones may cause inconsistencies in the code, which is prone to error. Managing thousands of clones manually would be a difficult task. A clone management system can help manage clones and find patterns of how clones evolve during the evolution of a software system. In this research, we propose a framework for constructing and visualizing clone genealogies with change patterns (e.g., inconsistent changes), bug information, developer information and several other important metrics in a software system. Based on the framework we design and build an interactive prototype for a multi-touch surface (e.g., an iPad). The prototype uses a variety of techniques to support understanding clone genealogies, including: identifying and providing a compact overview of the clone genealogies along with their key characteristics; providing interactive navigation of genealogies, cloned source code and the differences between clone fragments; providing the ability to filter and organize genealogies based on their properties; providing a feature for annotating clone fragments with comments to aid future review; and providing the ability to contact developers from within the system to find out more information about specific clones. To investigate the suitability of the framework and prototype for investigating and managing cloned code, we elicit feedback from practicing researchers and developers, and we conduct two empirical studies: a detailed investigation into the evolution of function clones and a detailed investigation into how clones contribute to bugs. In both empirical studies we are able to use the prototype to quickly investigate the cloned source code to gain insights into clone use. We believe that the clone management system and the findings will play an important role in future studies and in managing code clones in software systems.

Acknowledgements

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisor Dr. Kevin A. Schneider for his constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without his support and guidance, this work would have been impossible.

I would like to thank Dr. Chanchal K. Roy for his support and inspiration.

I would like to thank all of the members of Software Research Lab for their supporting me in hours of my need. Specially, I would like to thank Mohammad Asif Ashraf Khan, Muhammad Asaduzzaman, Minhaz Fahim Zibran, Ripon Kumar Saha, Md. Sharif Uddin, Md. Saidur Rahman, Khalid Billah, and Manishankar Mondal.

I would like to thank David Flatla from *the interaction lab* for helping me with his knowledge.

I am also grateful to Department of Computer Science, the University of Saskatchewan for their generous support through scholarship, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me in one way or another along the way. In particular I would like to thank Gwen Lancaster, Maureen Desjardins, and Heather Webb.

I express my gratefulness to my family members and relatives especially, my mother Sreelekha Saha, my father Dilip Saha, my brother Ripon Saha, my sister-in-law Rimpa Saha, my uncle Nil Ratan Saha, my aunt Mukty Saha, my friends Subrato Sarker, Anindya Das, Muhammad Izabul Khaled, Jasim Ahmed, Raju Saha and my cousins Rajesh Sikder, Pranab Kumar Hira, Prokash Kumar Hira, who did not get the share of my time that they deserved.

I would like to thank Aparna Saha for supporting and understanding me throughout the time.

I would like to thanks my friend Khadija Rasul for being with me in every good or bad decision. I would also like to thank Shomoyita Jamal and Eishita Farjana for treating me like family.

Invariably, acknowledgements always miss someone important. For those that I have not listed explicitly, thank you for being a part of this thesis and helping me grow as a person and a researcher.

I dedicate this thesis to my beloved mother Sreelekha Saha, whose selfless support and inspiration has always been with me at each and every step of my life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the software industry, maintaining existing software is inevitable. *Software maintenance* can be defined as the modification of a software product after delivery to improve performance, and other attributes, to fix bugs and to add features to better serve its purposes. Previous studies show that software maintenance can cost up to 80% of total effort[5]. To reduce maintenance cost, researchers are trying to improve tools that can be useful in software maintenance for detecting and reducing attributes that may hamper maintenance activities. It is believed that identical or similar code fragments in source code has an impact on software maintenance. Similar or identical code fragments are referred to as code clones.

Code cloning is a common practice in software development. Clones may be introduced into a software system by copying and pasting code fragments or may occur inadvertently during development and maintenance. Two or more code fragments that are identical or similar, and may have differences in comments or layout form a *Type-1* or *exact clone class*. Two or more clone fragments form a *Type-2 clone class* if they also have differences in the names of identifiers. In a *Type-3 clone class*, some lines can be added to or deleted from the clone fragments. Previous studies have shown that systems contain duplicated source code in amounts ranging from 5-15% of the code base [105] to as high as 50% [99]. Some researchers argue that the existence of similar or identical code fragments causes extra effort in maintenance activities [66], [70]. Clones are also considered a 'bad smell' in some studies [10], [60], [37]. For example, if a code fragment is buggy, all other fragments copied from it may replicate the same bug silently. Inconsistent changes to cloned code is frequent and may lead to severe unexpected behaviour [60]. On the other hand, some researchers show evidence that code clones have positive [70], [116] consequences for maintenance activities.

After a decade of active research, researchers are still arguing whether clones are good or not. As it is practically impossible to remove all clones [70], researchers agree that it is important to understand the evolution of clones for managing a system's clones properly. Therefore, we need to concentrate on managing clones efficiently and effectively. However, our experience shows that researchers and developers are not interested in all of the clone genealogies in a software system. Thus, a number of studies have been conducted to find patterns of code clone evolution to understand them more easily. This helps to focus on interesting clones. Researchers have already proposed some approaches for extracting clone genealogies. However, studies in the evolution of clones are mostly limited to Type-1 and Type-2 clones, but there are more Type-3 clones than Type-1 and Type-2 clones [104]. A software system can have thousands of code

clones that evolve across revisions. Thus, a genealogy extractor may extract thousands of clone genealogies. Mostly, they produce textual output, and it is difficult to find clone genealogies of interest from a large textual output. A visualization tool could help better understand clone genealogies. We propose a framework for extracting and visualizing clone genealogies that would help find clone genealogy patterns in less time and with less effort. The more patterns we can identify, the better we will be able to manage clones in software systems. In this research, we focus on the following problems in particular:

1. Since, researchers agree that we need to manage clones, we need a framework for extracting clone genealogies in software systems and for finding patterns of how clone classes evolve during the evolution of a software system.

2. A software system can have thousands of clone classes, thus a clone genealogy extractor can extract thousands of clone genealogies. Therefore, we need a tool that can find interesting genealogies and help us to better understand the evolution of code clones.

3. To better manage clones we need to study the evolution of code clones in different software systems so that we can find patterns.

1.1 Thesis Statement

In this research, we propose a framework for extracting and visualizing clone genealogies in a software system, which we use to build a prototype for a multi-touch surface and use to elicit feedback from practicing researchers and developers. Both the framework and the prototype help us to efficiently find clone patterns reducing the investment in time and effort, which in turn helps us to manage clones. To validate the usefulness of the framework and the prototype, we conduct two empirical studies and represent our findings by answering a number of research questions that requires a detailed investigation of the clones supported by the prototype.

1.2 Contribution

Our research opens up opportunities for studying clone evolution from a broader perspective. Our contributions are as follows.

1. **Clone Genealogy Extraction and Visualization Framework.** We present a framework for extracting and visualizing software clone genealogies. We consider Type-1, Type-2 as well as Type-3 clone classes as we know that there are more Type-3 clones than Type-1 and Type-2 clones [104]. Unlike other genealogy constructors [109], the framework is used to not only construct clone genealogies, but is also used to calculate several metrics (e.g., lifetime) and retrieve other information (e.g., buggy genealogies) that will help us to better understand a clone genealogy as well as making refactoring decisions. Furthermore, since the framework incorporates a visualization model that shows how the

information should be represented for better understanding clone genealogies, it can be used to build a visualization tool to visualize clone genealogies.

2. **Clone Genealogy Visualization Prototype.** To take advantage of our framework, we designed a user interface in accordance with the visualization model. We take into account several factors such as colors, space, and the organization of the interface. For choosing colors, we give preferences to those colors that are perceivable by people with common color vision deficiencies and people with normal vision. Then, we use the framework and the user interface to build a prototype for a multi-touch surface to visualize the evolution of code clones and get feedback from practicing developers and researchers. We have built the prototype for the iPad because of its portability, display quality, and gesture recognition capabilities. To the best of our knowledge, we are the first to introduce a prototype for visualizing clone genealogies on a multi-touch surface. Finally, we conduct structured and semi-structured interviews with practicing researchers as well as developers, and present their comments and feedback. We also compare the features we provided and the features experts expected. We have seen that we have implemented most of the features they expected. Furthermore, we addressed some other information that is important.

3. **Empirical Study on Function Clone Categories.** We extended the framework in order to investigate function clones in Java open source software systems. Researchers have conducted studies for finding patterns of Type-1, Type-2 and/or Types-3 clone genealogies. However, we further classified function clone classes (cf., Section 5.2) into five categories based on the return type and parameters of functions and analyzed their behaviour during the evolution of a software system. For example, if a clone class contains function clones only with no return type and no parameters, we call that clone class a FCType-1 clone class. Finally, we represent the findings by answering four research questions. First, we investigate which categories of function clones developers mostly create and how they live. We find that developers have the tendency to create FCType-2 function clones; however, they also create a significant number of FCType-4 function clones. We also find that there is about 53% to 93% of long lived FCType-2 genealogies and 51% to 82% of FCType-4 long lived genealogies in the subject systems. Second, we investigate which categories of function clones to look at. We conclude that FCType-2 and FCType-4 need extra attention while managing function clones in a software system. Third, we investigate how consistently the long lived function clone genealogies changed in the software systems and we find that only 1.28% to 21.72% of the total long lived clone genealogies changed consistently. Fourth, we investigate if function clones change over time. We find that they changed to another category and about 60% to 75% of the changed clone genealogies converted to FCType-2.

4. **Empirical Study on Bugs and Clone Genealogies.** Since bug fixing is an important part of software maintenance and our framework is able to find buggy clone genealogies, we were interested in how clones are related to bugs in open source software systems. We investigate three Java open source

systems to see how clones are related to bugs, and how buggy clones were managed during the evolution of a software system. We also perform statistical analysis to see whether there is a relationship between the growth of buggy clone groups and non-buggy clone groups over time. We classify clone classes into three categories based on the number of clone fragments, and investigate if there is any group of clone classes mostly involved in bugs. We also manually investigated randomly chosen buggy clone classes using the prototype. Finally, we represent the findings by answering four research questions. First, we investigate the extent buggy clone classes are related to bugs. We find that there is as low as 40% chances that there will be no buggy clone classes in a subject system. Second, we investigate how buggy clone classes are managed during the evolution and we find that more than 70% of buggy clone classes are changed inconsistently. Our manual investigation showed that in most cases those inconsistent changes either reproduced the same bug or created another bug. We also show that developers are not capable of remembering all the clones. Third, we investigate if there is a relationship between the growth of non buggy clone classes and buggy clone classes because the number of clones in a subject system increases over time [81]. Our statistical analysis shows that there is no strong relationship between them. Fourth, we find generally which category of buggy clone clone classes contribute to bugs. We show that 'Small' and 'Medium' categories of buggy clone classes exist more than that of the 'Big' category. Alternatively, we say that most of the buggy clone classes contain 2 to 10 clone fragments.

1.3 Summary

In this chapter, we discussed our motivation, research problems, and our contributions. The remaining chapters are organized as follows. In Chapter 2, we discuss background and related research. In Chapter 3, we describe our framework for constructing and visualizing clone genealogies. Chapter 4 presents our prototype for a multi-touch surface to help better understand the evolution of code clones. Chapter 5 describes an empirical study we conducted using the prototype to investigate how function clones evolve over time. Chapter 6 describes a second empirical study we conducted using the prototype to investigate how clone classes are related to bugs and how buggy clone classes were managed in software system over time. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, we provide background and discuss work related to our research, including a discussion of: code clones, the reasons for code cloning, drawbacks of code cloning, state-of-the-art tools and techniques for detecting clones, and code clone evolution. We also present research related to the evolution of code clones, and tools for visualizing code clones in software systems.

2.1 Code Clones

Code clones are similar or identical code fragments, often created for reusing source code by copying and pasting. Sometimes, clones are created accidentally, because of developing the same concept in different places [4]. Code clones can be classified as a *clone pair*, which consists of two code clones, and as a *clone class*, which consists of two or more code clones. There are four types of clone classes based on the degree of textual, syntactic, and semantic similarity among clone fragments [100], [106]. They can be described as follows:

Type-1: All clone fragments in a Type-1 clone class are identical to each other, but may have differences in comments or layout. A Type-1 clone class may also be referred to as an *exact* clone class. Table 2.1 shows an example of a Type-1 clone class, where the first two clone fragments are identical, but the third clone fragment has a comment.

Type-2: All clone fragments in a Type-2 clone class are similar, but may have differences in identifiers, literals, layout, and comments. Table 2.1 shows an example of a Type-2 clone class, in which the function names are different in each of the clone fragments.

Type-3: In a Type-3 clone class, statements can be added, modified, and/or deleted in the copied fragments in addition to variations in identifiers, literals, layout, and comments. Both Type-2 and Type-3 clone classes are known as *near-miss* clone classes. In Table 2.1, an example of a Type-3 clone class shows that a line has been added to fragment 2.

Type-4: In a Type-4 clone class two or more of the clone fragments are functionally the same, but are structurally different. From Table 2.1, all fragments of the Type-4 clone class perform the same computation but fragment 2 computes the result recursively unlike the others.

Table 2.1: Examples of Types of Clone Classes

Types	Clone Class		
	Fragment 1	**Fragment 2**	**Fragment 3**

Type-1	`int foo (int n) {`
	`  int a = 0;`
	`  for(int i=0;i<n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo (int n) {`
		`  int a=0;`
		`  for(int i=0;i<n;i++){`
		`  a=a+i;`
		`  }`
		`  return a;`
		`}`
			`int foo (int n) {`
			`  int a=0;//initialize`
			`  for(int i=0;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

Type-2	`int foo (int n) {`
	`  int a=0;`
	`  for(int i=0;i<n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo1 (int m) {`
		`  int a=0;`
		`  for(int i=0;i<n;i++){`
		`  a=a+i;`
		`  }`
		`  return a;`
		`}`
			`int foo2 (int n) {`
			`  int a=0;//initialize`
			`  for(int i=0;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

Type-3	`int foo (int n) {`
	`  int a=0;`
	`  for(int i=0;i<n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo1 (int n) {`
		`  int a=0;`
		`  for(int i=0;i<n;i++){`
		`  a=a+i;`
		`  }`
		`  a=a*10;`
		`  return a;`
		`}`
			`int foo2 (int n) {`
			`  int a=0;//initialize`
			`  for(int i=0;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

Type-4	`int foo (int n) {`
	`  int a=0;`
	`  for(int i=1;i<=n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo (int n) {`
		`  if(n==0)`
		`  return 0;`
		`  else`
		`  return n+foo(n-1);`
		`}`
			`int foo2 (int n) {`
			`  int a=0;//initialize`
			`  for(int i=1;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

2.1.1 Clones in Software Systems

Previous studies have shown that there is significant amounts of code clones in various software systems, depending on their domain and origin [52], [81], [70]. Baker [9] found that in large systems between 13% - 20% of source code can be cloned code, Lague et al. [78] reported that between 6.4% and 7.5% of functions were cloned in the systems they studied, and Baxter et al. [14] reported that 12.7% of code in a large software system was cloned. Mayrand et al. [90] estimated that industrial source code contains 5% - 20% duplicated code, and Kapser and Godfrey [67] have reported that as much as 10% - 15% of source code of a large system was cloned. In one object-oriented COBOL system, the rate of duplicated code was found to be even higher, at 50% [32]. Summarizing the studies, we can say that researchers have found code clones ranging from 5% to 20% to as high as 50% in different subject systems.

2.1.2 Reasons for Code Cloning

There are several reasons for code cloning, such as faster development, to keep software clean and so on. Code clone studies [9], [14], [21], [57], [63], [69], [90] have identified many reasons for code cloning. The reasons for

code cloning has been classified into the following four main categories by Roy and Cordy [105]:

Development Strategy: Developers often copy and paste code fragments for implementing the same functionality in a software system. For example, the ports for external inputs of a subsystem are similar in functionality. Sometimes developers reuse a similar solution. For example, to create a driver for a hardware family, a driver of a similar hardware family can be reused with a slight modification. Clones may also be produced when merging two software systems with similar functionality and when auto generating code.

Maintenance Benefits: To achieve maintenance benefits, sometimes clones are introduced intentionally. For example, it may be less risky to reuse well trusted code that has already been tested several times rather than developing new code. Clones may also be introduced to keep a software architecture clean as the clones can then evolve independently.

Overcoming Underlying Limitations: The underlying limitations of programming languages and developers is another reason for code cloning. Sometimes it is easier to manage code clones than it is to write reusable code. Developers often introduce clones because of: difficulties in understanding a large system, time constraints, incorrectly measuring developer productivity by code output, a lack of knowledge, and a lack of ownership of the code being reused.

Cloning by Accident: Sometimes clones are created accidentally. For example, two developers can implement the same functionality in the same way, or programmers may unintentionally repeat a common solution for similar kinds of problems.

2.1.3 Drawbacks of Code Cloning

In the previous section, we discussed various reasons behind code cloning. However, there are some negative impacts of code cloning on software systems. Sometimes quality, re-usability, and maintainability of software systems may be affected adversely by some cloned code snippets [60]. In this section, we will describe some of the consequences of code clones stated in the literature to explain why at times we need to find and remove them.

Impact on Modification

Often software developers create clones so that they can evolve independently without affecting each other. However, this may cause additional time and effort to understand the existing clone implementation, therefore it may become difficult to add new functionality to a system, or even to change existing functionality [57], [90]. In addition, if a cloned code is buggy by any chance, all other clone fragments should be checked when making changes to fix the bug(s). It also multiplies the amount of work while maintaining or enhancing cloned code snippets [90], [95].

Bug Propagation

Cloned code may cause the introduction of new bugs. For example, if a developer copies and pastes a code snippet with or without modification without knowing about the bugs in it, then the bugs will propagate with all the copied code fragments in the system. On the other hand, a developer may forget to propagate changes to all the clone fragments, which can produce a bug. It may increase the probability of bugs significantly in a software system [56], [82].

Understanding Effort

To maintain cloned code, maintenance engineers are required to have knowledge about all of the existing clones, whether systems are small or large, which is time consuming and cumbersome for large systems as clones can be dispersed among several files or directories. To understand the differences between all clone fragments, they need to examine all of the clones in a software system [56].

Design Issue

Clones have some negative impacts on software design as well. They may be the cause of lack of a good inheritance structure or poor abstraction. Clones are not always reusable in future projects. As a result they may lower software qualities such as readability and changeability [95].

Resource Requirement

Code clones are nothing but multiple occurrences of a code snippet, which inflates the size of a software system. The size of a software system may not be important in all domains, but in some domains (e.g., telecommunication switches or compact devices), it may require a hardware upgrade with a software upgrade. In addition, larger software systems often require increased compilation time. Furthermore, larger systems typically have an increased financial impact.

2.1.4 Clone Detection Technique

Since code clones have both positive and negative consequences on software development and maintenance activities, the software engineering research community showed their interests on clone detection techniques and they proposed different techniques to detect clones. Clone detection is also important for assisting software developers in understanding code clones in software systems, especially in large software systems. The approaches are mainly classified as textual approaches, lexical approaches, tree-based/syntactic approaches, graph-based approaches, and metric-based approaches. In this section we briefly describe these approaches.

Textual Approach

Textual approaches compare source code as texts with little or no transformation or normalization prior to the actual comparison. In most cases, these approaches use raw source code for detecting clones. Therefore, these approaches are independent of the programming languages, and even work for the source code which is not compilable. Johnson [57] is the first who introduced a text-based clone detection approach that uses fingerprints on substrings of the source code to find clones in the source code. Manber [88] also uses fingerprints, based on the subsequence marked by leading keywords, to identify similar files.

Marcus and Maletic [89] used latent semantic indexing (LSI) technique to find a high level concept clones (e.g., abstract data types (ADTs)) in the source code. This approach limits its comparison in comments and identifiers instead of the entire source code.

NiCad [101] is a text-based approach that takes the advantage of the tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering. Thus, it eliminates the conventional drawbacks of the textual approach, and has high precision and recall. Recently, Uddin et al. [118], [117] improved a modified version of NiCad by incorporating a text similarity measurement technique called simhash [18], which was found to be effective in fast detection of both exact and near-miss clones.

Lexical Approach

Lexical approaches transform source code into a sequence of lexical *tokens* similar to compilers, the tokens are scanned for duplicated subsequences, and the corresponding source code is returned as clones. These are also called token-based approaches. The lexical approaches overcome the limitation of textual approaches for finding clones with minor code changes such as formatting, spacing, and renaming. Dup [9], CCFinder [63], iClones [41] are some of the examples of token-based clone detectors.

Tree-Based Approach

This approach transforms source code to a parse tree or an abstract syntax tree, and then tree-matching algorithms are used to find the similar subtrees. If a similar subtree, which is a clone, is found, the corresponding source code is returned as a clone. It is independent of programming style (e.g., formatting), therefore, in some cases it is better than the text-based and token-based approaches. It has higher precision compared to the textual and lexical approaches. However, this approach is dependent on programming languages, and requires syntactically correct program. Furthermore, the time complexity of tree-based approaches is higher than that of the textual and lexical approaches. Baxter et al.'s CloneDr [14], Jiang et al.'s Deckard [52], Koschke et al.'s cpdetector [73] are some of the examples of tree-based clone detection tools.

Graph-based Approach

A graph-based approach represents the source code of a program as a program dependency graph (PDG). In a PDG, nodes are statements and predicate expressions, and edges represent controls and data dependencies among the vertices [35]. Therefore, in a PDG representation, source code is independent of the sequence of statements, and thus this approach is more robust for simple modifications of the code clones such as reordering of lines. Then, clones can be searched by finding isomorphic subgraphs [74]. The main limitations of PDG-based approaches are the same as the limitations of tree-based approaches. This approach is program language dependent, requires syntactically correct program, and has high time complexity.

Metrics-based Approach

A metrics-based approach calculates a number of metrics for code fragments at a certain level of granularity. Functions/methods, classes, or any syntactic units can be the level of granularity. The algorithms compute different metric values, then compare the metric values to find clones in the source code. Generally, most of the metrics-based clone detection tools are language dependent as the calculation of many metrics are language dependent. There are a number of clone detection tools that detect clones using metrics-based algorithms. Mayrand et al. [90] used several metrics such as names, layout, expressions, and simple control flow of the functions to identify functions with similar metric values as code clones. Davey et al. [24] detect exact, parametrized, and near-miss clones by first computing certain features of code blocks and then training neural networks to find the similar blocks based on the features. Metrics-based approaches have also been applied to find duplicate web pages or finding clones in web documents [17], [87].

Furthermore, some clone detection techniques use a combination of syntactic and semantic characteristics [80] to detect clones in source code. Clone detection is not limited to source code only. Sæbjørnsen et al. [107] proposed a practical clone detection algorithm for binary executables. Davis and Godfrey [25], [26] introduced a tool that can compile C, C++, and Java to assembler, and then perform clone detection on the resulting stream of assembler instructions contained within functions. Deissenboeck et al. [27], [28] presented an approach for the automatic detection of clones in large models. Nguyen et al. [96] and Pham et al. [97] also proposed some techniques for finding clones in MATLAB/Simulink models. There are also some other techniques to detect clones in other software artefacts. Domann et al. [29] proposed an approach for detecting clones in requirement specifications. Later Juergens et al. [62] applied the approach to study clones in real world requirement specifications. Liu et al. [83] and Storrle et al.[113] proposed techniques for detecting clones in the UML sequence diagrams and models respectively. A survey by Roy and Cordy [105] presents more details about each approach.

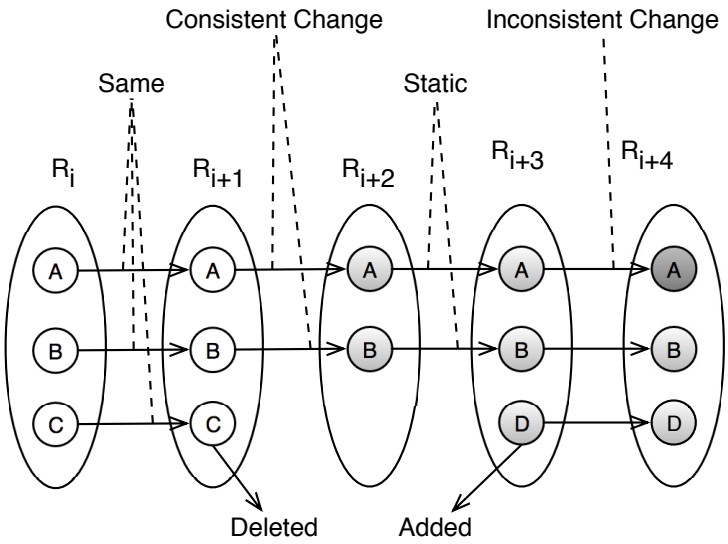


Figure 2.1: A clone genealogy with different changes

2.2 The Evolution of Code Clones

Developers may make changes to code clones during the evolution of a software system, which may affect the system positively or negatively. Studying the evolution of code clones helps us to understand how code clones change over the life-time of a software system and how those changes may affect a software system. In this section we discuss code clone genealogy models and studies of code clone evolution.

2.2.1 Code Clones Genealogy Model

A clone genealogy describes how a clone fragment changes over versions with respect to other fragments in a clone class. Kim et al. [56] were the first to define a clone genealogy model. They also identified six change patterns based on the changes to code snippets and the number of clone fragments in the same clone class in two consecutive versions. We adapted their model of clone genealogy in this thesis. Now we briefly discuss the terminology relevant to our clone evolution model, various change patterns, and genealogies.

- **Revision**: As this thesis is concerned with the evolution of code clones, the work involves more than one revision. A revision can be defined as a snapshot of the source code of a software system as stored in a software repository along with some important information, such as: all changes made to the source code, developer information, timestamps and so on.

- **Clone Lineage**: A clone lineage is a directed acyclic graph that describes the evolution history of a clone class from the beginning to the final version of the software system.

- **Clone Genealogy**: A clone genealogy is a single clone lineage or a set of clone lineages that originate from the same clone class.

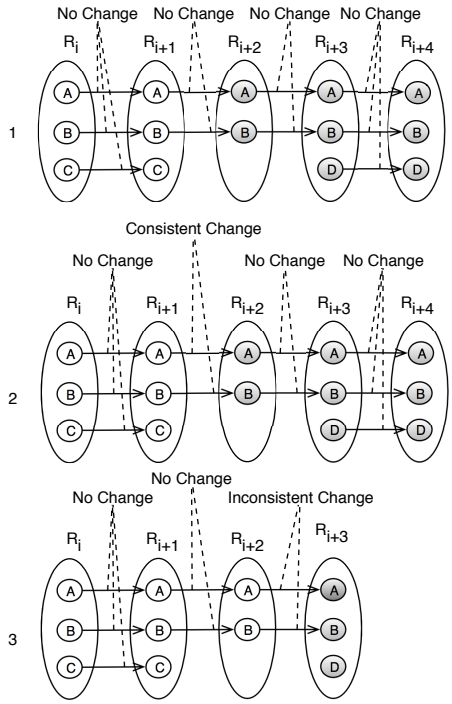


Figure 2.2: Different types of clone genealogies

Change Patterns

Let CC_i be a clone class in revision R_i, which is mapped to a clone class CC_{i+1} in revision R_{i+1} by a clone genealogy extractor. Now the change patterns can be described as follows:

- **Same:** The clone fragments in CC_i are present in CC_{i+1} and no additional clone fragment has been added in CC_{i+1}.

- **Add:** One or more clone fragments are added to CC_{i+1} that were not present in CC_i.

- **Delete/Subtract:** One or more clone fragments of CC_i do not appear in CC_{i+1}

- **Static:** The clone fragments in CC_{i+1} that were part of CC_i, have not changed.

- **Consistent Changes:** All of the fragments in CC_i have been changed consistently, thus, all the fragments are again part of CC_{i+1} in R_i. However, a clone class may disappear after being changed consistently, if fragments become smaller than the minimum clone length of the clone detection tool.

- **Inconsistent Changes:** All clone fragments in CC_i have not been changed consistently. Here we should note that as lines can be added to or deleted from Type- 3 clones, all the clone fragments of a particular clone class could still form the same clone class in the next revision even if one or more fragments of that class have been changed inconsistently. The dissimilarity between clone fragments in a clone class depends on the heuristics or similarity threshold of clone detection tools.

Types of Clone Genealogies

Clone genealogies in software systems can be categorized as follows:

- **Static Genealogy (SG)**: In a static genealogy, the clone fragments in a clone class propagate through subsequent revisions without any modification during the evolution of a clone class. In Figure 2.2, genealogy 1 represents a static genealogy.
- **Consistently Changed Genealogy (CCG)**: A consistently changed genealogy can have any consistent change patterns but cannot have any inconsistent change patterns. In Figure 2.2, genealogy 2 represents a consistently changed genealogy as this genealogy consistently changed between R_{i+1} and R_{i+2}.
- **Inconsistently Changed Genealogy (ICG)**: A clone genealogy can be referred to as an inconsistently changed genealogy if the clone class associated with the genealogy changed inconsistently during its evolution. In Figure 2.2, genealogy 3 represents an inconsistently changed genealogy as there is an inconsistent change between R_{i+2} and R_{i+3}.
- **Dead Genealogies**: A genealogy is called a dead genealogy if its clone class disappears before reaching the final revision. In Figure 2.2, the genealogy 3 represents a dead genealogy since it disappears in R_{i+4}.
- **Alive Genealogies**: A genealogy is called an alive genealogy if the associated clone class is still evolving and thus exist in the final revision. In Figure 2.2, both genealogy 1 and genealogy 2 represent alive genealogies because they still exist in R_{i+4}.

2.2.2 Clone Genealogy Extraction

To understand clone genealogies, we need to extract clone genealogies from multiple revisions of a software system. There are several approaches proposed by researchers. Kim et al. [70] detected clones in each version of a program and then they mapped consecutive versions to construct clone genealogies. In some studies [8], [75], clones are detected in versions of interest and the detected clones are tracked in subsequent versions to understand their evolution. Another study [15] constructed clone genealogies using a combination of the first two approaches. Some studies [41], [1] mapped the clone fragments during clone detection using change information between versions. In a study by Saha et al.[109], they mapped clone fragments between two consecutive versions using the longest common subsequence count (LCSC) algorithm and then they constructed clone genealogies from the mapped data.

2.2.3 Study of Code Clone Evolution

To better understand the evolution of clone genealogies, several studies have been conducted in the last decade. Still there are disagreements whether the clones are harmful or not. It is also true that researchers

agree that we need to manage clones to take full advantage of using clones. In this section, we will discuss studies of code clone evolution.

Clone Coverage

Laguë et al.[78] conducted a study to ensure there is a need for a clone detection tool in software development by analyzing six versions of a large telecommunication system. They investigated how clones evolve such as addition, modification, and/or deletion of clones during the evolution of the system across versions. They found that although a significant number of clones were removed during the evolution of the system, the overall number of clones increased over time. On the other hand, they did not investigated how clone fragments changed with respect to other clone fragments in a clone class and how the changes affected the system.

Antoniol et al.[6] proposed a model of cloning to monitor and predict the evolution of code clones in a software system using time series. They validated their model with several versions of a medium scale software system (mSQL), and concluded that time series can predict the clone percentages of subsequent releases with an average error rate below 4%. Another study of Antoniol et al. [7] investigated the Linux Kernel and found that most of the clones are clustered into the subsystems, but few clone classes were distributed across the subsystems. The overall number of clones over versions was stable. Godfrey and Tu [43] also found similar results and concluded that cloning is a common and steady practice in the Linux kernel.

After investigating the Linux Kernel and FreeBSD, Li et al. [81] found that the rate of cloning increased gradually over time for both subject systems. During the evolution period, which was about 10 years, the cloning rate increased by 5% for the Linux Kernel. A similar observation was also found for FreeBSD. They extended their investigation to the module level and found that the rate for a few modules, `drivers` and `arch` in the Linux Kernel and `sys` in FreeBSD was actually significantly higher than the entire system. Finally, they concluded that this phenomena was due to the extensive support of the Linux Kernel for many similar device drivers during that period.

Zibran et al. [121] performed a large empirical study to understand the proportion and evolution of near-miss clones in evolving software systems. They used a regression analysis technique to predict clone density in future versions of software systems. They also performed quantitative analysis and manual investigation on over 1636 releases of 18 software systems. They concluded that the evolution of clone density is significantly affected by programming languages but a little bit affected by a system's size. The number of both exact and near-miss clone fragments increases with the growth of functions in a system showing a very strong correlation between them.

Change Patterns of Code Clones

Kim et al. [70] were the first to map clones across versions of a program to see their evolution. They defined a model of clone genealogy (cf., Section 2.2.1) including some meaningful patterns. After investigating two Java

subject systems, they reported that on average 36%-38% of total clones changed consistently, many clones are volatile in the software systems, and some clones are long lived. They also reported that an immediate refactoring of short-lived clones is not require and some long-lived clones are not locally refactorable due to the limitations of the underlying programming languages.

Aversano et al. [8] further divided the inconsistent change patterns into two groups: independent evolution and late propagation. If the clone fragments of a clone class changed inconsistently once and evolve independently afterwards across revisions it is called an *independent evolution*. And, if the clone fragments of a clone class changed inconsistently and later, at some point in their evolution, they changed again to synchronize it is called *late propagation*. They conducted an empirical study with those change patterns and how bug fixing activities take place during the evolution. They extracted change information from the CVS repositories of the subject systems, then they investigated clone detection results to see how clones evolved during the evolution. Kim et al. [70] also manually investigated all the genealogies where changes took place in different Modification Transactions (MT) and clone fragments were from different files. They concluded that the majority of clones are always maintained consistently. They also found that when clones are not changed consistently, they mostly evolve independently. Thummalapenta et al. [116] found similar results in an extended study.

Krinke [75] investigated five Java systems to see the changes that occurred frequently. Like Aversano et al. [8], he also extracted change information from software repositories. In his study, he showed that half of the changes to the clone classes were changed inconsistently. Another study of Göde and Koschke [41] showed that clones are rarely changed during their lifetime and if they are changed, they tended to be changed inconsistently.

In a recent study, Göde and Harder [39] conducted a case study on three open source software systems to analyze different combinations of consecutive change patterns during the evolution of clones to find if there is any unwanted inconsistent changes. Based on this case study, they reported that there are many clones that were changed more than once and there were few instances of unintentional inconsistent changes. But, they did not report any relationship between the consecutive change patterns and such unwanted inconsistencies.

Stability of Cloned Code

Krinke conducted a case study [76] on five open source software systems with 200 revisions to analyze the stability of the cloned code. He observed that if the dominating factor of deletions is eliminated, it can generally be concluded that the cloned code is more stable than the non-cloned code, and thus requires less maintenance effort. In another study [77], he takes the advantage of the subversion system (SVN) to analyze how frequently cloned code and non-cloned code changes in a subject system. He investigates exact clones and shows that the cloned code is older than the non-cloned code in the subject systems, which again supports that cloned code is more stable than non-cloned code.

Göde and Harder [42] replicated and extended Krinke's study [76] using their incremental clone detection

technique, iClones [41] to validate the outcome of the study. They supported Krinke by assessing the cloned code to be more stable than the non-cloned code in general, and the non cloned code is more stable with respect to deletions. Their study also reveals that larger clones are more stable with respect to changes while more unstable with respect to additions. They also reported that generally the reason behind the deletion of the cloned code was to perform restructuring and cleanup activities instead.

In a recent study, Hotta et al. [51], measured frequencies of modifications of the cloned code and the non-cloned code to analyze the impact of the clones on software maintenance. They concluded that the modification frequency of the non-cloned code is higher than that of the cloned code, which also implies that the cloned code is more stable than the non-cloned code.

Mondal et al. [92], [94], [93] conducted several empirical studies using three methods associated with the respective set of stability measurement metrics using twelve diverse subject systems covering 3 programming languages to validate the studies [76], [77], [51]. They considered three types of clones. They concluded that the clones in Java and C systems are not as stable as the clones in C# systems. Furthermore, a systems development strategy might play a key role in defining its comparative code stability scenarios.

In order to investigate the relationship between code clones and maintenance effort, Lozano et al. [86] compared measures of maintenance effort on methods with clones against those without clones. Although, in the study they showed that the functions with clones changed more often than the functions without clones. However, in a later study, Lozano and Wermelinger [84] investigated four open source Java software systems and showed that some methods with clones significantly increase the maintenance effort. Finally, they concluded that there is no systematic relation between the clones and such maintenance effort increase.

Change Anomalies

Aversano et al. [8] investigated bug fixing changes of code clones and they found 17 bug fixes that were involved with code clones. They found that there were four consistent changes and six changes were classified as independent evolution as the bug was corrected in some of the clones. They also found seven changes as a result of late propagation.

In order to examine the characteristics of late propagation in more detail, recently Barbour et al. [13] conducted an empirical study using two open source Java systems, where they considered only Type-1 and Type-2 clones. They classified late propagation into eight categories based on the following modifications of clone pairs:

1. **Clones Modified in Diverging Change.** Either one or both clones can be modified independently during divergence. For example, clone A or clone B or both can be modified during the first inconsistent change of a late propagation genealogy.

2. **Clones Modified During Period of Divergence.** Either one, both or neither clone can be changed during the period of divergence. For example, clone A or clone B or neither of them can be changed before the re-synchronization change.

Table 2.2: Classification of Late Propagation

LP Type	Clone Pair	Clones Modified in Diverging Change	Clones Modified During Period of Divergence	Clones Modified During Re-synchronizing Change
LP1	$< A, B >$	A	A	B
LP2	$< A, B >$	A	A, B	B
LP3	$< A, B >$	A	A	A, B
LP4	$< A, B >$	A	A, B	A
LP5	$< A, B >$	A	A, B	A, B
LP6	$< A, B >$	A, B	A, B	A *or* B
LP7	$< A, B >$	A, B	A, B	A, B
LP8	$< A, B >$	A	A	A

3. **Clones Modified During Re-synchronizing Change.** Either one or both clones are modified to re-synchronize a clone pair. For example, clone A or clone B or both of them can be changed to re-synchronize the clone pair. Table 2.2 shows all categories of late propagation. They concluded that late propagation genealogies are more prone to fault than other clone genealogies, especially LP8 and LP7, are riskier than other types of late propagation genealogies.

Bakota et al. [10] investigated suspicious changes to identify potential problems. They defined four distinct clone smells: Vanished Clone Instance (VCI), Occurring Clone Instance (OCI), Moving Clone Instance (MCI), and Migrating Clone Instance (MGCI). While the VCI and OCI are same as the Delete and Add change pattern respectively as described in Section 2.2.1. If a clone fragment moves from one clone class to another clone class, then they classified this as the MCI. And, if the moved clone class, moves back to the previous clone class in a later version, then they classified this as an MCGI.

Bettenburg et al. [15] conducted an empirical study to analyze the effect of inconsistent changes on software quality at the release level. They analyzed two open source software systems and found that only 1% to 3% of inconsistent changes to the clones introduced software defects.

Researchers have conducted studies to investigate the effects of software systems in order to ensure good software quality. Juergens et al. [60] detected inconsistent clones in software systems by their tools and used manual annotation by developers to determine faults in inconsistent clones. They concluded that unintentionally made inconsistent clones are more likely to contain defects. They did not provide a statistical test of significance. Jiang et al. [53] proposed an approach to detect clone related bugs based on contextual similarities. Then based on contextual difference, they suggested whether a possible bug is lurking. Thummalapenta et al. [116] studied clone maintenance and evolution in software systems. They showed that clones were consistently propagated when needed. They did not directly relate the results with buggy clones. Śliwerski et al. [112] studied changes to source code that induce fixes. Instead of finding bug inducing changes we investigate changes between the bug fixing revision and the intermediate revision. Rahman et al. [98] showed that more than 80% of buggy code contained no cloned code, but they did not show whether most of the clone classes are buggy or not. They did not consider clone classes. It might happen that buggy clones are the only clones in a revision. In which case, clones would be considered to be extremely bad.

Recently, Saha et al. [110] conducted an exploratory study to understand the evolution of Type-3 clones using six open source software systems. They showed that the absolute number of consistently changed Type-3 clone classes is greater than the number of Type-1 and Type-2 clone classes and they have a lifespan

similar to that of the Type-1 and Type-2 clones. They also showed that some of the Type-1 and Type-2 clones converts to the Type-3 during their evolution, thus it is important to manage the Type-3 clones properly.

Some clones increase the maintenance effort and others do not. It is still unclear which clones are real threats to a systems quality and need to be taken care of. Göde et al. [40] analyzed the evolution of code clones in mature software projects and showed that clones are rarely changed and that the number of unintentional inconsistent changes to clones is small. We thus have to carefully select the clones to be managed to avoid unnecessary effort managing clones that have no risk potential.

2.3 Clone Management

As we discussed earlier in Section 2.1.2, developers often create clones intentionally because of several benefits. Although, it would be safer not to have clones or we could refactor all of them, however, it is not feasible to refactor or remove all clone from a software system. Therefore, to take maximum advantage of code cloning while overcoming all threats, we need to manage clones properly. In this section we will discuss several approaches for managing code clones.

2.3.1 Clone Prevention

The main goal of clone prevention is to prevent creation of the code clones instead of detecting and removing clones after the development phase. Lague et al. [78] described two ways of how a clone detection tool could help to avoid the clones in the software development process. One way is called *preventive control*. In this way, a clone detection tool confirms whether a new function is a cloned code fragment or not, and if it is a cloned code fragment, then it can only be used for specific reasons. If the system architect is not convinced by the provided reason, necessary actions must be taken to reuse the original function. Another way is called *problem mining* where all changes submitted to the central source code repository are monitored. If a clone is found, then developers are informed of these clones so that they can take the necessary action.

2.3.2 Clone Correction

In this management technique, suspicious clone fragments are refactored to reduce risk factors from those clone fragments and to clean up the code to support better understanding. Finding and removing uninteresting clones is an important task for better software maintenance. There are several studies on clone refactoring. In this section, we discuss corrective clone management techniques.

The simplest method of clone refactoring is extracting and replacing exact code clones by a new function created from shared code of the clone fragments. This method can be defined as *extract method* [34], [47], [61], [72]. Fanta and Rajlich [34] removed functions and class clones from industrial object-oriented systems using an automated restructuring tools. Higo et al. [47] proposed an approach for refactoring clones from object oriented software using existing refactoring patterns, especially *Extract Method* and *Pull Up Method*.

They also implemented a refactoring tools with their method. Juillerat and Hirsbrunner [61] also used the extract method refactoring for the Java language and they detected clones with an AST-based approach. Komondoor and Horwitz developed a semantics preserving procedure extraction algorithm that works on PDG-based clones [72]. Balazinska et al. [12] used design patterns to refactor code clones from Java subject systems. Unlike other studies, Kim et al. [69] anticipated that developers may be inclined to refactor larger and frequently copied fragments.

Tairas and Gray [114] conducted two separate studies on an open source software system to investigate if developers refactor sub-clones properly or not. After the investigation, they found a number of instances of sub-clone refactoring where only part of the clone ranges are actually refactored. They suggested that sub-clone refactoring facilities should be incorporated in a clone management system.

Göde [38] is the first who tried to remove clones retrospectively from a maintainers' point of view. He conducted a case study on four subject systems to understand how developers deal with clones in the real world. He found many instances of deliberate clone removal. He also noticed that most of the clones were refactored through method extraction.

Choi et al. [19] performed a study using various combination of metrics to extract clones. The main goal was to find a precise combination of metrics based on developer feedback. However, this method has two potential threats to validity and they are 1) only one system and one developer were involved in their study, 2) the study was dependent on only three specific metrics, and there was no metric that considered the change history.

Zibran and Roy [120] presented a refactoring effort model, and proposed a constraint programming approach for conflict-aware optimal scheduling of code clone refactoring to maximize benefit and minimize refactoring effort.

2.3.3 Compensative Clone management

To minimize the software maintenance effort, several techniques and tool supports have been introduced. However, there are still some clones that are not worthy to refactor. This approach tries to facilitate the evolution of this group of clones. Miller et al. [91] proposed an approach of simultaneous editing that helps developers to make the same changes to all clone fragments of a given clone class at the same time. Therefore, it helps preventing unintentional inconsistent changes. Duala-Ekoko and Robillard [30] have proposed a tool called CloneTracker that can notify developers when developers intend to change a clone fragment, and offers simultaneous editing.

2.4 Clone Visualization

Most of the clone detection tools report the basic information of clones such as file name, line numbers, start line, end line etc. in the form of clone pairs and/or clone classes in a textual format. However, clones in

a software system may differ in several contexts such as clone type, degree of similarity, granularity, size, etc. Insufficient clone information makes it difficult to understand in depth the clones in a software system. Sound visualizations of clones would help better understanding clones in a system. In this section, we will discuss some of the visualization techniques that have been proposed in the literature.

Visualizing clones using a scatter plot [20] is a popular technique. This technique presents clones in the form of two dimensional charts where software units are listed on both axes [9], [32], [99], [119]. If two units have clones in common, a dot is used to represent the information of a clone pair as a diagonal line segment with different granularities of software units. Scatter plot techniques is useful to select and view clones, as well as zoom in on regions of the plot. However, the scalability issue limits its usability for large systems. Higo et al. [49] introduced an enhanced scatter plot approach that overcomes this limitation. They showed that an enhanced scatter plot is also good in understanding the state of the clones for different versions of a software system. It also filters out uninteresting clones before the result is displayed.

Johnson [58] used Hasse diagrams for visualizing clones between files. A Hasse diagram consists of nodes and edges where clones and its associated files are represented by a node and the relation between clones are represented as an edge. The height of a node in the graph is determined by its size, large files or code segments are towards the bottom, and similar segments of code are towards the top. Later on he proposed to navigate the web of files and clone classes via hyper-linked web pages [59]. The hyperlink functionality of HTML allows users to jump freely between source files related to clone fragments in a clone class, however, although it is very easy to navigate, it does not allow a user to see the states of code clones over the system.

In addition to scatter-plots, Gemini [50] uses the output of CCFinder to provide visualization through metrics graphs and file similarity tables. It allows users to browse clone pairs and clone classes individually.

Rieger et al. [99] used Lanza and Ducasses polymetric views [79] to visualize code clones. Polymetric views help investigate code clones at different levels of abstraction, thus provide more information about cloning in a software system. They also visualized clone relationships in order to easily find different units of interest.

Kapser et al. [65] developed a tool, CLICS, to visualize clones that uses the output of CCFinder and a taxonomy of clone types [64] for visualization. It is able to visualize clone information with structures in source files. It also supports query-based visualization that helps users find clones of interest easily. They did not use a scatter plot because of its limited scalability.

Tairas et al. [115] introduced an Eclipse plug-in for displaying the results of CloneDR. Their approach extends AJDT visualizer[1], which is different than a scatter plot for visualizing clones. The integration of CloneDR with Eclipse allows the tools to take advantage of the rich environment of the IDE, which offers frameworks for a configuration wizard, views and editor connections. A user can determine the type of configuration for the clone detection procedure. Then, the plugin call CloneDR and produce a text file containing its clone detection results. The results are parsed and send back to Eclipse views to produce a

[1]http://www.eclipse.org/ajdt

graphical representation of the results.

Adar and Kim [3] were the first to analyze the evolution of code clones visually through SoftGUESS, a system for clone evolution exploration. SoftGUESS is developed on top of GUESS [2], the graph exploration system, that models the evolution of a software using graphs. They mainly focused on structural dependencies and clone evolution in conjunction with containment relationships. However, they did not focus on change patterns, fragments changes, type changes, etc.

A study by Zhen et al. [55] proposed a technique for visualizing cohesion and coupling between architectural subsystems. Jiang and Hassan [54] have also proposed a framework for understanding clone information in large software systems. They use a data mining technique framework that mines clone information from the clone candidates produced by CCFinder. Another study of Ball and Eick [33] described a set of views (matrix view, cityscape view, bar and pie charts, data sheets and network view) to hint at changes in software using visual metaphors.

CYCLONE [45], a multi-perspective tool for clone evolution analysis, offers five different views for analyzing code clones. It uses simple rectangles and circles to visualize clone genealogies where each circle represents a clone fragment arranged in a set of rows that represents a particular version and each rectangle represents a clone class that contains all of the clone fragments that belong to it. They used lines and colors to represent the evolution of clone fragments. However, it takes a large amount of space and produces a high volume of data that limit its usefulness.

2.5 Summary

In this chapter, we discussed code clones, the reasons for cloning, clone management, etc. and we defined the terminology that we use in this thesis. We also briefly discussed related research.

Chapter 3

A Framework for Constructing and Visualizing Clone Genealogies in a Software System

3.1 Motivation

In the previous chapter, we discussed the advantages and disadvantages of code clones. It is clear that we need to take full advantage of code clones for better software development eliminating the parts that may cause problems for software maintenance. Therefore, managing code clones is an important task during software maintenance, and the study of the evolution of code clones makes it easier to manage code clones in software systems. A software system may contain thousands of clones; thus, there could be thousands of clone genealogies. And, of course, it is not worthwhile to find the genealogies of interest manually as it may take a lot of time. We need a framework that can be used to extract clone genealogies with useful information to understand clone genealogies, and can support the construction of a visualization tool.

Researchers have used different approaches to map clones across versions. In [10], [70], [108], clones are detected in all versions of interest and then clones are mapped between consecutive versions based on heuristics. However, this approach has quadratic time complexities [44]. Moreover, if a clone changes significantly and goes beyond the similarity threshold, this approach may not map the clone further. In [8], they detected clones from the first version of selected versions and then the clones are mapped between consecutive versions based on change logs provided by source code repositories such as *svn*. However, this approach will fail to track the clones that appeared in later versions. In [41], [1], clones are mapped during clone detection based on source code changes between revisions. This approach is incremental and fast enough to detect and map clones across a given set of revisions, but if a new revision is added for mapping, it will run the whole process again, which is lot more time consuming. Another study [15] used the combination of the first two approaches. In [109], they proposed an automatic framework for extracting exact (Type-1) and near-miss clone (Type-2, and Type-3) genealogies. Their approach is also incremental and fast. Furthermore, it does not run from a start revision like [41] for mapping a given set of revisions. However, they ignored several useful metrics such as how dissimilar are the clone fragments in a clone classes in a genealogy and if a genealogy is related to a bug. They also did not provide any model to visualize the data.

In this chapter, we propose a framework for extracting exact and near-miss clone genealogies with change

patterns discussed in Section 2.2.1 and visualizing clone genealogies efficiently to better understand clone genealogies in software systems. Unlike [109], the framework automatically identifies fault fixing clone genealogies to support investigating bugs due to clones, determines the dissimilarity value of a clone class in a genealogy to find out accidental clone classes easily, incorporates developers' information so that they can be contacted if necessary and used to investigate if there is a relation among developers and clones, and several other important metrics.

The rest of the chapter is organized as follows: Section 3.2 describes the additional terms we use in this chapter; Section 3.3 describes the framework; Section 3.4 shows a comparison between our framework and a framework similar to our framework; and finally, Section 3.5 summarizes this chapter.

3.2 Terminology

Since, clone terminology varies among papers we define the following terms to make our use clear to readers. We already discussed code clones, clone genealogies, and change patterns in Chapter 2. In this chapter, we will use a few additional terms as follows.

Buggy Clone Class/Group: The clone classes that are changed to fix faults are called buggy clone groups.

Intermediate Revision: In this thesis, we will call the immediately previous revision of a fault fixing revision as an intermediate revision. For example, in a software system, if a bug is fixed in revision r, then the revision $r-1$ is an intermediate revision.

Fault Fixing Revision: During software development, each commit to a software repository is a revision. The revision committed to fixing faults is called a fault fixing revision.

3.3 Framework

Our main goal is not only to build a framework for constructing and visualizing genealogies, but is also to build a framework that can be used to automatically identify important information such as change patterns, bug fixing clone genealogies, and developer details; because, the more information we identify, the more we will be able to find patterns and the more patterns we discover, the better we will be able to manage clones in software systems. The framework performs the following activities in order to achieve the goal i) process selected revisions of a software system, ii) process clone classes and calculate metrics for further investigation, iii) map clone classes between consecutive revisions, iv) incrementally construct clone genealogies using the clone maps, v) construct clone genealogies, vi) process clone genealogies to retrieve information that is important to better understand clone genealogies, and vii) organize data according to the visualization model. In this section, we discuss the activities of the framework.

Table 3.1: NiCad 2.9 Settings

Option	Value
Granularity	Function
Minimum Clone Length	5 LOC
Identifier Renaming	Consistent Renaming
Dissimilarity Threshold	30%

3.3.1 Process Software Revisions

In this section, we discuss some preliminary processing necessary for constructing clone genealogies. First, we take all the revisions of a software system that are selected for genealogy construction from a software repository. Then, we process all revisions in two independent steps. In step 1, we detect clones from all revisions, and in step 2 we mine the software repository. Figure 3.1 shows how we process software revisions.

Clone Detection

To detect clones from all the revisions in step 1, we needed a clone detector. We gave preferences to those tools that provide clone class information, are effective for detecting near-miss clones, and have high precision and recall. Based on our criteria, we chose NiCad [22], [23] to detect clones from subject systems because it has been shown to be effective for detecting near-miss clones with high precision and recall [103], [104], [101], [102]. We have carefully chosen clone detector settings (as shown in Table 3.1) to detect clones from the subject system. We set the granularity to 'function' because we are not interested in the clones that start from a function and end in another function. It will also reduce the number of false positive clones. Selection of minimum length of clones (in terms of the number of lines) is important because the precision will increase with the minimum length of clones while the recall will decrease with it. We take 5 LOC as minimum size of clone because it is enough to get good precision and good recall. Another important setting is dissimilarity (UPI) threshold value for selecting Type-3 clones. It refers to how dissimilar the clone fragments can be in the same clone class. We allowed 30% dissimilarity among the clone fragments in a clone group or class.

After setting up NiCad, we detect clones for all the selected revisions of a subject system. NiCad provides clones in XML and HTML file format. It provides very basic information about each clone class such as clone fragments, number of clone fragments. During the clone detection process, it extracts all functions and all consistently renamed functions in a software system in separate XML files.

Process Software Repository Logs

In this step, we store repository logs for all revisions of a software system. A log in a software repository contains important information such as the date of a commit, developer's name, developer's email, and commit message. We will use the commit messages later to identify fault fixing revisions (cf., Section 3.3.5), and developers' information to contact developers if needed (cf., Section 3.3.6).

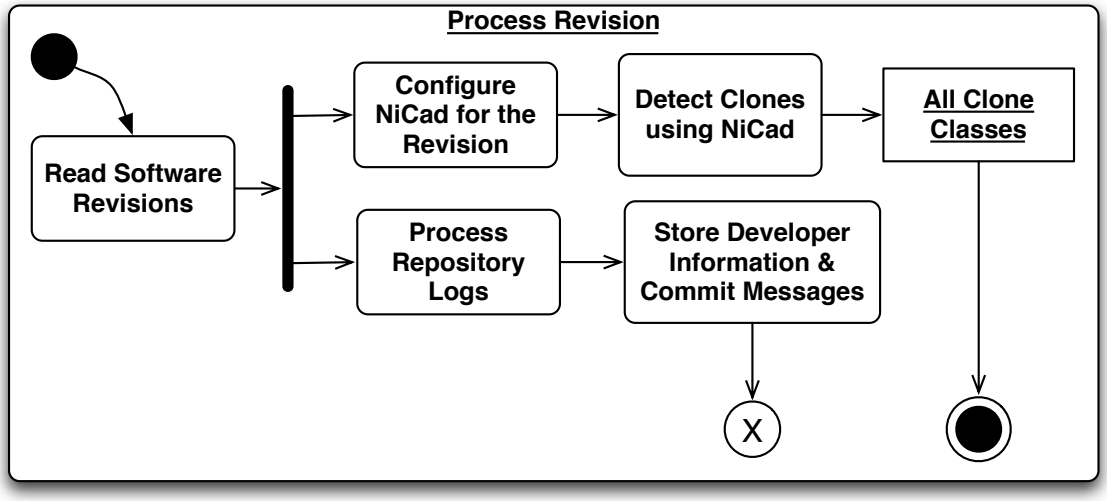


Figure 3.1: Process each revision

3.3.2 Process Clone Classes

After detecting all clones from all revisions, we further process the clones. First, we categorize clone classes by lines of cloned code (LOCC). Second, we categorize them by clone types. And, third we take a measurement of how clone fragments are dissimilar in a clone class. After having all the information, we update clone classes with all the new information and finally, we store them for future use. The steps are described below in detail.

Categorize Clone Classes By LOCC

To categorize clone classes by LOCC, we take the number of lines of each clone fragment. Then, we take the maximum number of lines to categorize a clone class. We categorize clone classes into three categories based on the maximum LOCC, and they are Small(S), Medium(M), and Large(L). Table 3.2 represents how we classify all the clone classes in a software revision. This is a default setting, but it is customizable as researchers may be interested in different values.

Categorize Clone Classes By Types

After categorizing by LOCC, we classify clones of each revision by their types. As we discussed earlier in Section 2.1, there are four types of clone classes based on their degree of textual, syntactical, and semantic similarities among clone fragments. In this study, we consider only three types of code clones, and they are Type-1, Type-2, and Type-3. We iterate through the clone classes of each revision to categorize them. If we find that all fragments in a clone class are identical, we mark that clone class as a Type-1 clone class and if we find additions or deletions of lines in any fragments in a clone class, we mark that clone class as a Type-3 clone class; otherwise, we mark any other clone classes as Type-2 clone classes. After categorizing all the

Table 3.2: Category of Clone Classes based on LOCC

Category	LOCC	Letter
Small	< 10	S
Medium	10 - 20	M
Large	> 20	L

clone classes, we update the clone classes with their type.

Calculate Dissimilarity

In Type-2 and Type-3 clone classes, clone fragments can be dissimilar. In this study, we take the measurement of the maximum dissimilarity value for each clone class of a revision. The main objective of calculating the dissimilarity value is to find clones easily based on their similarity or dissimilarity so that developers can decide whether they should refactor most similar clones or not. To calculate dissimilarity, we use the longest common subsequence count (LCSC) algorithm because previous studies [71], [109] have successfully used this algorithm to compare function names and clone fragments. We use Equation 3.1 to get a dissimilarity value for each clone class in a software revision. We update all the clone classes in a revision with the dissimilarity values.

$$Dissimilarity = 1 - \{\frac{LCSC_{AB}}{|A|} + \frac{LCSC_{AB}}{|B|}\}/2 \tag{3.1}$$

Calculate Distribution Size

Distribution size of a clone class can be defined as the total number of files in which clone fragments are distributed. This information is important to know while modifying a clone class to fix some bug because if changes do not propagate to all the clones properly, it may reproduce the same bug or create new bugs. We calculate the distribution size for all the clone classes in a revision of a software system. To calculate distribution size we check the file paths for each clone in a clone class and take the total number of unique file paths.

3.3.3 Map Clone Classes

For constructing clone genealogies, we need to map clone classes across revisions. We map clone classes of two consecutive revisions at a time. To map clone classes between two consecutive revisions, we follow the same approach proposed by Saha et al. [109]. First, we map functions between two consecutive revisions and then we map clones using the mapping information. After mapping all clones, we map clone classes. In this section, we briefly discuss how we map clone classes between two consecutive revisions.

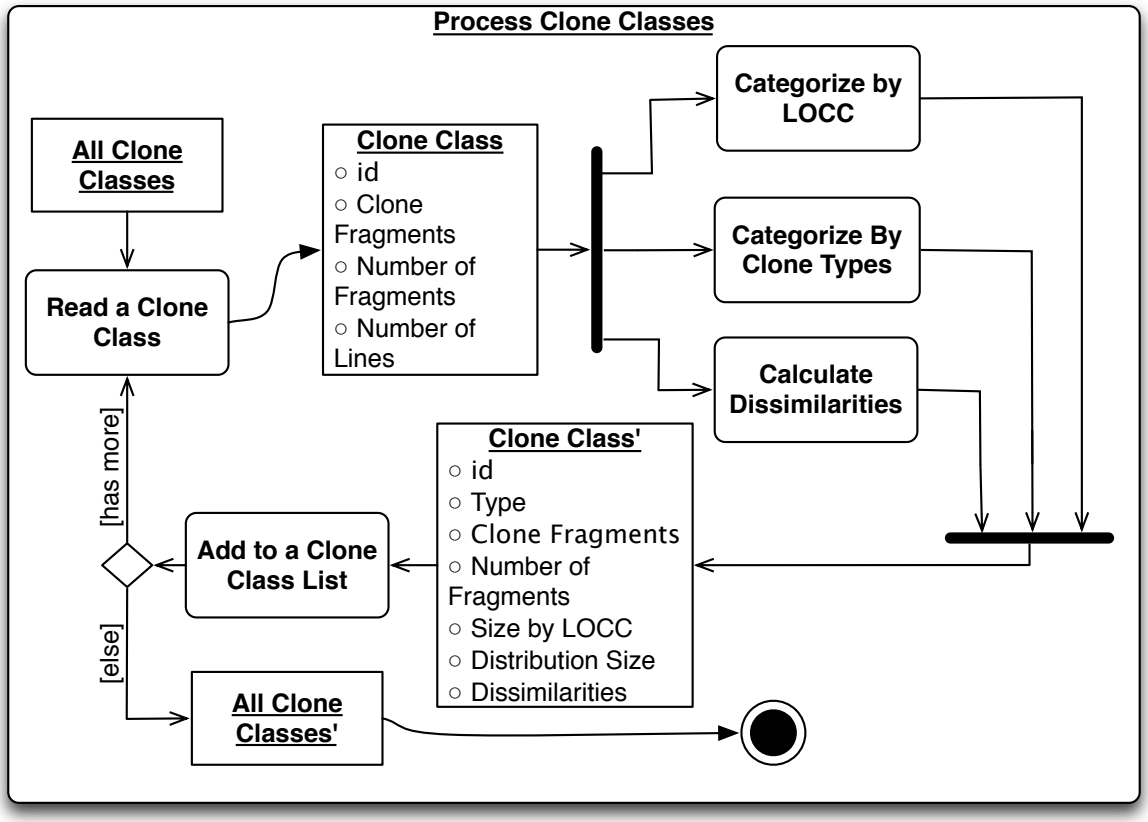


Figure 3.2: Process clone classes

Map Functions

First, we map functions across two consecutive revisions r_i and r_{i+1}. We consider the signature of functions along with their class name and full path. However, in practice some functions could be renamed, could move to different files or directories. When function names remain the same, we find if it occurs once in r_i and once in r_{i+1}, it is considered the same function without considering any further information. On the other hand, if two or more functions exist having the same name in either one or both versions, then we check the location and signature of the functions. When the functions are renamed, we use the function name and body to map functions across two consecutive versions. We use LCSC to find the origin of a function. We use equation 3.2 to calculate LCSC similarity between two fragments A and B where $|A|$ and $|B|$ are the number of elements in A and B respectively. The LCSC similarity metric returns value between 0 and 1 where 1 means exactly equal and 0 means no similarity.

$$Similarity = \{\frac{LCSC_{AB}}{|A|} + \frac{LCSC_{AB}}{|B|}\}/2 \tag{3.2}$$

Map Clones

We map clones from r_i to r_{i+1} using the function maps. Because, in this study, each clone fragment is a function as we set NiCad's granularity parameter to 'function'.

As we have mapping data between clone fragments of r_i and r_{i+1}, so, to map a clone class cc_i in r_i, we find mapped clone fragments in r_{i+1} for all clone fragments of cc_i in r_i. Then, we find the clone class cc_i in r_{i+1} using the mapped clone fragments in r_{i+1}. However, while dealing with Type-3 clones, due to extensive inconsistent changes, a clone class may split in the next revision. If cc_i of r_i split to cc_{ix}, cc_{iy}, and cc_{iz} in r_{i+1}, then we map cc_i of r_i to $\{cc_{ix}, cc_{iy}, cc_{iz}, ...\}$ of r_{i+1}.

Identify Changes

We automatically identify change patterns of each clone class of a subject system on the server. Detection of consistent and inconsistent change patterns is challenging for Type-3 clones. We use a multi-pass approach and identify consistent change and inconsistent change gradually. First, we identify the static clone classes and clone classes that have been split in the next revision. If a clone class split in the next revision, we consider this change as an inconsistent change. Because, they split due to extensive inconsistent change.

Second, we consider Type-1 (exact) and Type-3 (where modifications are limited to line additions and deletions but no variable renaming). We compute the differences between two mapped clone classes using the *diff* algorithm. If the difference between each clone pair is the same, then they are marked as a consistently changed clone class, otherwise as an inconsistently changed clone class. However, *diff* cannot detect reordered statements.

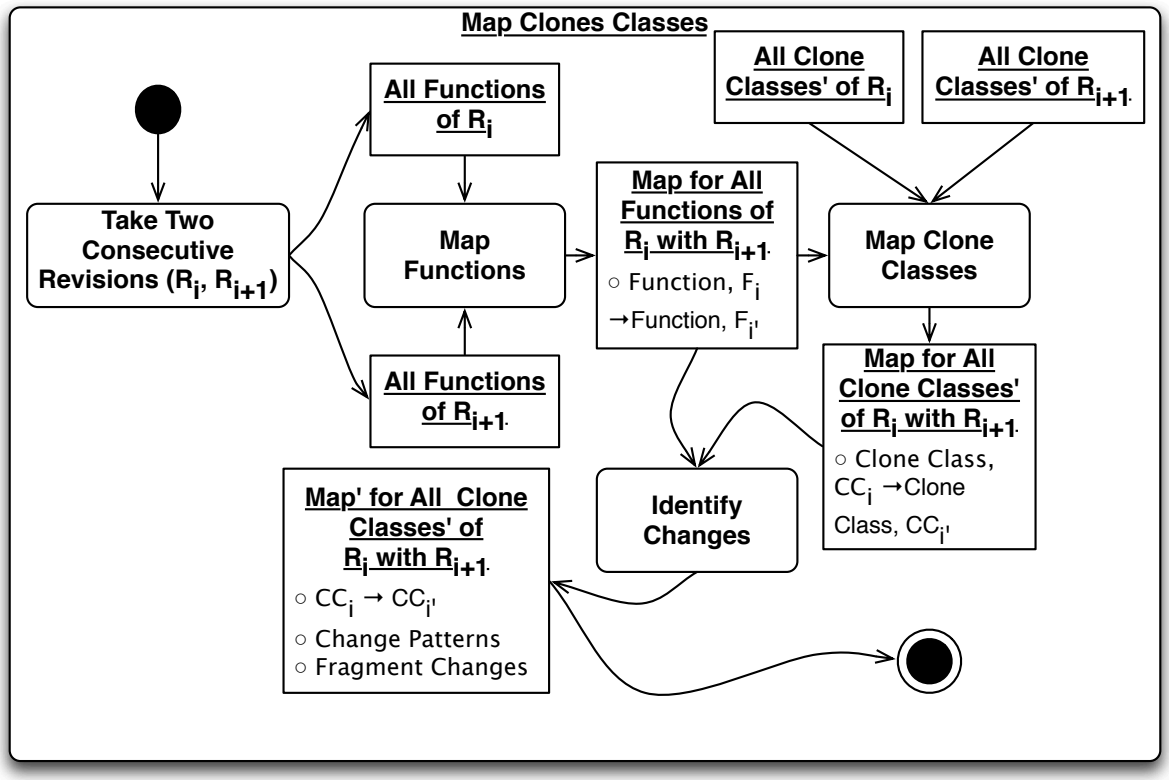


Figure 3.3: Map clone classes

Third, we consider Type-2 and Type-3 clones (with identifier renaming). We compare consistently renamed files (generated by NiCad). Then as before we calculate the differences using *diff*. If the difference is the same then it is consistently changed; otherwise it is inconsistently changed.

3.3.4 Construct Genealogy

Once we have the mapping between every two consecutive revisions, we have the genealogies for each two consecutive revisions. To construct genealogies across all the revisions, we concatenate all genealogies in each consecutive revisions. For example, if we have n revisions of a software system, then after mapping, we have genealogies of $\{R_1, R_2\}$, $\{R_2, R_3\}$, ... , and $\{R_{n-1}, R_n\}$. Then, to construct genealogies we combine them together, and they can be represented as $\{R_1, R_2, R_3, ..., R_n\}$. Then, we iterate through the genealogies for further investigation.

3.3.5 Process Genealogies

After mapping, we incrementally construct genealogies for the selected revisions of a subject system. After genealogy construction, we iterate through the genealogies and calculate important metrics such as lifetime of a genealogy, and genealogy type. Finally, we store all genealogies for future use. In this section, we discuss how we process genealogies. Figure 3.4 also describes graphically how we process clone genealogies.

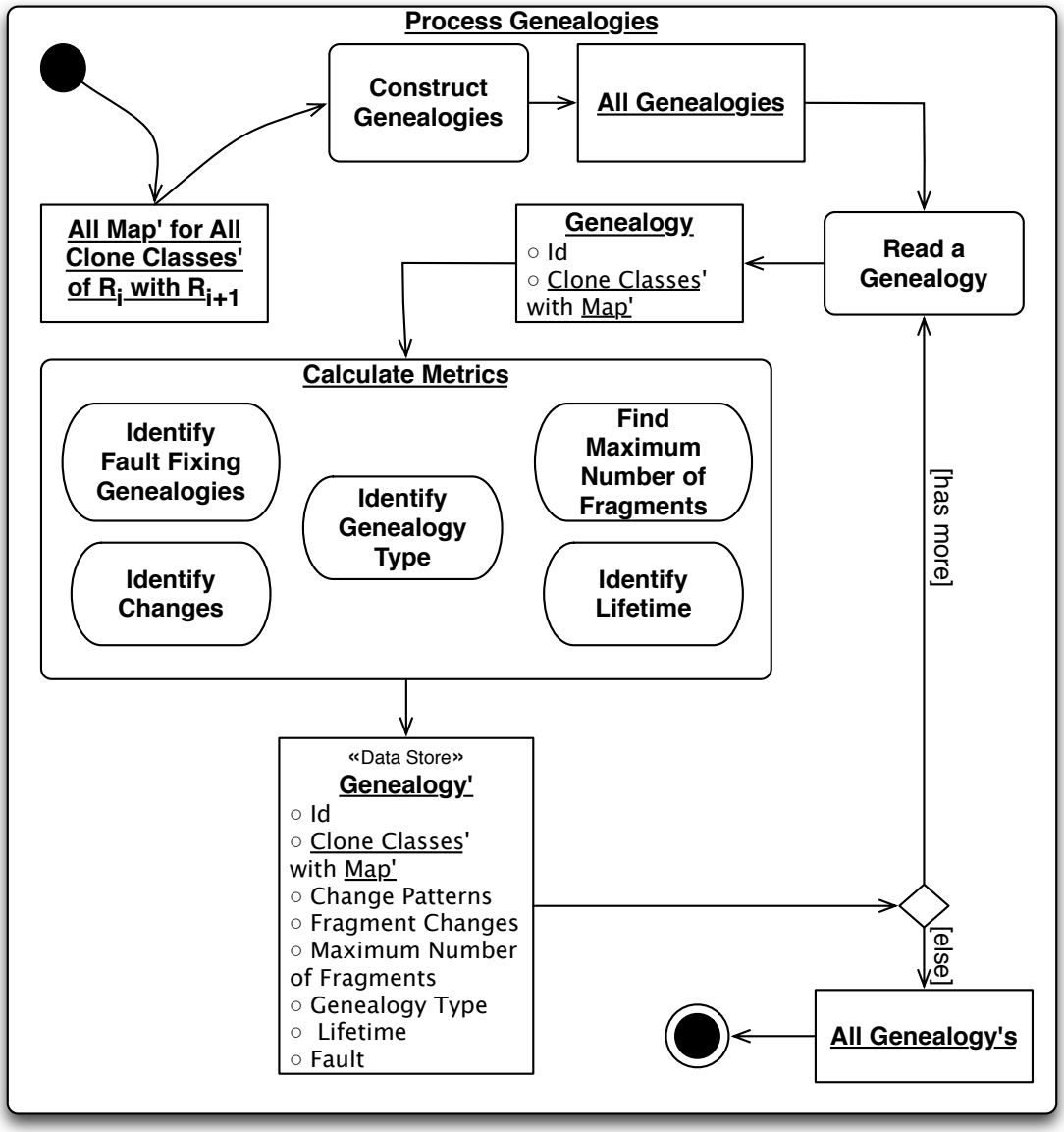


Figure 3.4: Process clone genealogies

To learn more about a clone genealogy, we calculate the size of a clone genealogy based on the maximum number of clone fragments in a clone class of a clone genealogy. This will help us to find genealogies of interest based on size. Then we calculate the lifetime (the number of revisions a clone class survives) of a genealogy, which will help us determine dead genealogies and alive genealogies. Then, we identify genealogy types so that we can find different types of genealogies easily and we can also find if a genealogy changes types due to inconsistent changes. We also identify changes to know how a clone class evolved during the evolution, and whether a clone genealogy is fault fixing or not. Figure 3.4 depicts the process for calculating metrics. In this section, we will describe the process in detail.

Calculate Genealogy Size

To calculate genealogy size, we iterate through a clone genealogy and take the maximum number of clone fragments in a clone class. This metric will help us find clone genealogies of those clone classes that have a large number of clone fragments.

Identify Lifetime

To identify lifetime, we take the revision where a clone class appeared and the revision when a clone class disappeared. Then, we take the difference to calculate the lifetime of a clone genealogy. In this process, we also identify whether a clone genealogy is an alive genealogy or a dead genealogy. We have already discussed alive genealogy and dead genealogy in Section 2.2.1.

Identify Genealogy Type

We check the type of each clone class in a genealogy to determine the type of the genealogy. For example, a genealogy of a Type-1 clone class is called a Type-1 genealogy. However, a clone class may change type during the evolution and in that case we determine the genealogy type using the type of the initial clone class. We also store the information if a clone class changes its type so that we can easily find genealogies of those clone classes that changed their type.

Identify Changes

We iterate thorough all genealogies to investigate how genealogies were changed. We have already discussed static genealogies, consistently changed genealogies, and inconsistently genealogies in Section 2.2.1. If a clone class propagates through revisions without any modification, we mark them as a *static genealogy*. If a clone genealogy has gone through consistent changes at some points, but never gone through inconsistent changes, then we mark the genealogy as a *consistently changed genealogy*. And, if a clone class evolves with some inconsistent changes, then we mark the genealogy as an *inconsistently changed genealogy*.

We also identify fragment changes (cf., Section 2.2.1) in a clone genealogy. If any fragments are added in a clone class during the evolution, we mark that clone class as *added*. Similarly, if we find any delete change

patterns we mark them as *delete*.

Identify Fault Fixing Genealogies

To identify fault fixing genealogies, first, we need to identify fault fixing revisions. We use commit messages that were stored while processing revisions of a software system (cf., Section 3.3.1), and the prior fault studies [111], [46] to identify fault fixing revisions. Then, we find intermediate revisions. For example, if revision r is a fault fixing revision, then the immediate revision $r-1$ is called an intermediate revision. The reason to choose intermediate revision is to approximate buggy clone. We are not interested in finding the origin of a buggy code fragment. We use the *diff* algorithm to see the changes made to fix bugs, and if any clone class is changed to fix bugs, we call that clone class a *Buggy Clone Class*. The genealogies of those clone classes that were changed to fix fault(s) are marked as fault fixing genealogies.

3.3.6 Model for Visualization

The aforementioned activities of the framework construct clone genealogies and store them in a database. These can be shown as text, but finding information from a large textual dataset will be troublesome. Furthermore, a clone genealogy contains important information that is necessary to consider when taking further action (e.g., refactoring). Figure 3.5 shows a class diagram with a data organization useful for visualizing clone genealogies to support understanding clone use in a software system. This model mainly focuses on presenting most the important data in each view to support making decisions for managing clones. Below we discuss more about each module in the class diagram below.

Clone Fragment

A clone fragment is a duplicated code fragment in a software system. A clone fragment can have the following information

1. **Start Line:** This is the line number where a clone fragment starts.
2. **End Line:** This is the line number where a clone fragment ends.
3. **Number of Lines:** This is the total number of cloned lines. We calculate the number of lines by using the simple following equation 3.3.

$$NumberofLines = StartLine - EndLine + 1; \tag{3.3}$$

4. **File Name:** This is the name of the file, in which a clone fragment is located.
5. **Absolute path:** This is the complete path of the file, in which a clone fragment is located.
6. **Source Code:** This is the cloned source code. We may need this for several analysis.

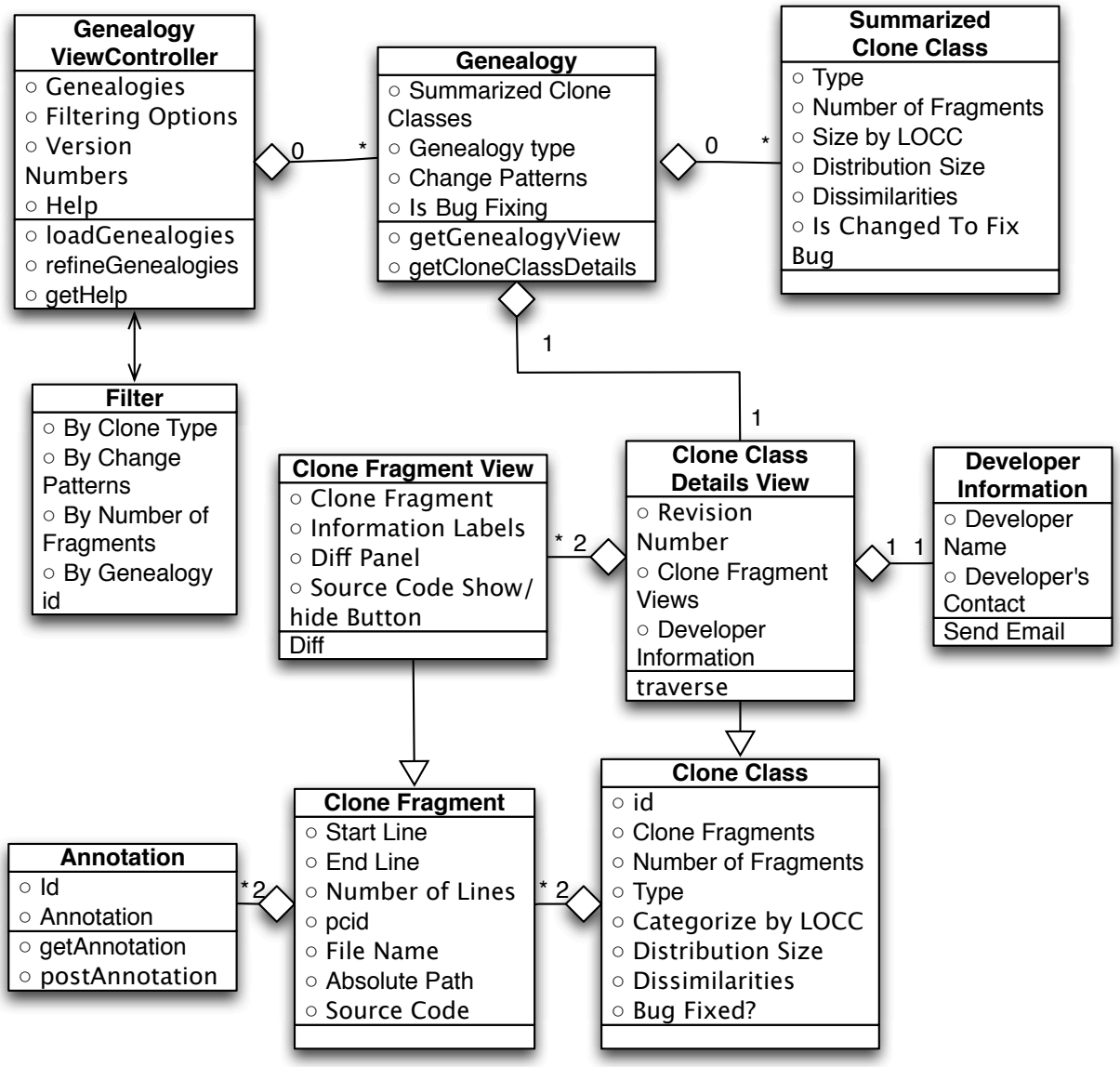


Figure 3.5: Basic class diagram for visualizing clones

Clone Fragment View

A *Clone Fragment View* visualizes a clone fragment in an organized way. It has the following attributes.

1. **Clone Fragment:** This is the clone fragment, which will be visualized.
2. **Information Labels:** These labels visualize all information except the source code of a clone fragment.
3. **Diff Panel:** This is a panel that contains options to facilitate users to see *diff* between fragments.
4. **Source Code Show/Hide Button:** We keep the source code separate from other information, because, source code can be large. A user should be able to see the source code if needed.

In this view, a *diff* algorithm should be implemented to show diff between two clone fragments if needed. A user also should be able to annotate each clone class.

Annotation

During investigation, developers and/or researchers might want to put some thoughts on a clone fragment. Therefore, we propose to have an option of annotation on each clone fragment view (cf., Section 3.3.6) so that they can annotate clone fragments and see it later when needed.

Clone Class

Two or more clone fragments together form a clone class. In Figure 3.5, the "Clone Class" class represents the attributes we consider representing a clone class. An instance of clone class contains all of its clone fragments with detail information. It contains other information we calculated in Section 3.3.2 such as clone type, clone size, distribution size, dissimilarities, and bug fixing information. Bug fixing information was collected while processing genealogies (cf., Section 3.3.5). It helps understand whether a clone class was modified to fix a fault or not.

Summarized Clone Class

When presenting a clone genealogy, it is useful to provide an overview instead of presenting all the attributes for each instance of a clone class. We call this overview of an instance of a clone class a *Summarized Clone Class*.

Genealogy

A clone genealogy describes how a clone class evolves during the evolution of a software system. Thus, the main idea for representing a clone genealogy is representing instances of a clone class with change patterns until it disappears. The attributes of a genealogy can be described as follows.

1. **Summarized Clone Class:** To give an overview to a user, we use a summarized clone class to represent an instance of a clone class in a genealogy.

2. **Genealogy Type:** We represent a genealogy with the genealogy type because it will help finding different types of genealogies easily. Furthermore, a clone class may change its type during its evolution; thus, it will help find those genealogies that have changed type.

3. **Change Patterns:** Since, a genealogy represents how a clone class changes during the evolution, we added change patterns (cf., Section 2.2.1) to represent the changes of a clone class across revisions.

4. **Is Bug Fixing:** A clone class may be changed to fix bugs during its evolution. We also mark those genealogies so that a user can find fault fixing clone genealogies easily.

Genealogy View Controller

A genealogy view controller visualizes clone genealogies of a subject system. A clone genealogy contains a set of instances of a clone clone class, change patterns of the clone class, bug information etc. as the genealogy represents how a clone class is evolving across the selected revisions until it disappears. A genealogy should provide users the opportunity to visit each clone class for further investigation. Since, a subject system can have thousand of genealogies, we also need some filtering options to find interesting genealogies. For example, if anyone wants to find Type-1 genealogies, it would be hard to look up all genealogies and find out Type-1 genealogies one by one. We discuss more about filtering options next.

Filtering

Filtering is an important part for finding clone genealogies of interest from thousands of clone genealogies. In this model, we propose four filtering options. They are given below.

1. **By Clone Type:** Using this option users will be able to find genealogies by the clone type. For example, if anyone wants to see only Type-2 and Type-3 genealogies, s/he can filter them using this option. It is easier than manually finding them.

2. **By Change Patterns:** Using this option users will be able to find a set of genealogies with change patterns (e.g., genealogies with inconsistent changes) easily.

3. **By Number of Fragments:** This option allows users to find clone genealogies by a range of the number of clone fragments. For example, a person is interested in those genealogies that have clone fragments between 10 to 20, s/he can find those clone genealogies easily using this option.

4. **By Genealogy Id:** This option allows users to select a number of clone genealogies of interest by their id for analyzing them together.

Developer Information

We already stored developer information while processing revisions in Section 3.3.1. We use the developer's name who committed the revision and his/her email address so that s/he can be contacted if necessary. This information also will be necessary, if anyone wants to find a relationship between developers and clones.

Clone Class Details View

A *Clone Class Details View* shows details of a clone class. A user comes to this view from a genealogy view controller whenever a user wants to see details of a clone class of a genealogy. A user can see the following information in this view.

1. **Revision Number:** When a user comes to this view from a genealogy view controller, it shows a clone class, but to keep track in which revision s/he is in, the revision number can be shown.

2. **Clone Fragment Views:** This view represents each clone fragment using a clone fragment view, which was already discussed in Section 3.3.6 so that the user can see all clone fragments together and perform *diff* operations staying in this view.

3. **Developer Information:** This view will display developer's information so that the user knows who committed this revision or who changed this clone class.

3.4 Comparison

In this section, we compare our framework with a recent similar study in which, Saha et al. [109] proposed a framework, gCad, for extracting and classifying near-miss clone genealogies. Table 3.3 shows a detailed comparison between our framework and the gCad Framework. The gCad only shows a clone class type when presenting a clone genealogy, whereas we present a clone class with type, dissimilarity, distribution size, size by LOC, and number of fragments which will help us to better understand a clone genealogy. They did not look for the genealogies that are buggy, and that were changed to fix a bug. However, we automatically mine repository logs to find fault fixing revisions and identify fault fixing genealogies so that we can investigate whether a clone class was changed consistently to fix a fault or not. Unlike gCad, we include developer information in this study so that they can be contacted if needed. Furthermore, we present a detailed visualization model, which clearly explains the data organization and views so that the models can be incorporated into a visualization tool, but they did not provide any visualization model. We represent the output using JSON format, which can be parsed easily for further analysis whereas they represent their output in simple text.

Table 3.3: Comparison with gCad

About	**Information**	**Our Framework**	**gCad**
Clone Class	No. of Clone Fragments	✓	×
	Lines of Code	✓	×
	Distribution	✓	×
	Type	✓	✓
	Dissimilarity	✓	×
	Changed to Fix Bug?	✓	×
Genealogy	Change Patterns	✓	✓
	Fragment Changes	✓	✓
	Genealogy Type	✓	×
	Type Changes	✓	×
	Life Status	✓	✓
	Bug Relation	✓	×
	Genealogy Size	✓	×
Other	Visualization Model	✓	×
	Developer Information	✓	×
	Data Format	JSON	Plain text

3.5 Summary

Since clones have both positive and negative aspects, we want to take maximum advantage of code clones eliminating the parts that may cause problems. To utilize code clones, we need to manage them properly. Therefore, we need a clone management system that can help us to manage clones in a software system. In this chapter, we propose a framework for extracting and visualizing clone genealogies in a software system. The framework uses several techniques to automatically identify important information such as change patterns, bug fixing clone genealogies, and developer's details to better understand the clone genealogies in software systems. It also incorporates a visualization model with a data organization so that our framework can be used for visualization tool development. Finally, we compare the framework with the gCad framework [109], and from the comparison, we show that unlike gCad, our framework is able to find important information such as bug information, and developer information.

Chapter 4

Clone Visualization: A New Experience with Multi-Touch Surfaces

In the previous chapter, we propose a framework for extracting and visualizing clone genealogies in a software system automatically identifying change patterns and several metrics that would help us better understand clone genealogies. In this chapter, we propose new user interface ideas for multi-touch surfaces to visualize clone genealogies in a software system. Then, we use our framework and user interfaces to build a prototype for a multi-touch surface and elicit feedback from practicing researchers and developers.

4.1 Motivation

The strategy of source code cloning is used for several purposes (e.g., faster development) despite the risks of using them. Cloned code may change during the evolution of a software system. Several studies indicated that we need to focus on managing code clones rather than trying to remove them. However, understanding the evolution of code clones manually is not an easy task, since a software system may have thousands of clone fragments. Furthermore, it has also found that cloned code fragments cause extra effort to maintenance activities [68], [70]. For example, there are two cloned code fragments in a system and later on, a bug is found for which one of those clone fragments is responsible, then we need to modify the clone fragment and propagate changes to the other clone fragment as well. Otherwise, the other clone fragment will remain buggy. Thus, better tool support can help understand the evolution of clones in a software system.

We believe that understanding the evolution of clones is an important part of maintaining clones in a software system. To better understand the evolution of code clones, we need better tool support. There are already some tools for mapping clones across consecutive versions or revisions of a software system [70], [10], [31], [41], [85], [1], but few of them visualize the evolution of code clones [3]. Most of the genealogy extractors produce textual data, and it is a cumbersome task to understand the evolution of code clones in a software system from a large amount of textual data. To support clone study, we are motivated to build a new tool that would help us in understanding the evolution of code clones efficiently. We are interested in exploring new user interface ideas that would allow us to present many useful clone metrics in a single view and to easily navigate to view clone details when necessary.

In this chapter, we propose new user interface ideas for visualizing the evolution of code clones in a

software system on multi-touch surfaces. We chose multi-touch surfaces in order to investigate gesture-based exploration of clone information. We also try to understand what information researchers generally look for, based on our framework discussed in Chapter 3. Then we designed the user interface in a way so that each view presents useful information. We use colors, text, and symbols to make the user interface informative. We choose colors for the interface very carefully so that people with the most common forms of color vision deficiencies (CVD) can see the user interface properly. In a genealogy, a developer or a researcher might want to traverse clone genealogies in a software system to see how the number of clone fragments in a clone class (a clone class contains two or more similar or identical code fragments) changed over time, which is very common. If s/he has to click or hover the mouse pointer on each clone class to see this information, then it adds to the overhead as the subject system may have thousands of clone genealogies. Therefore, we make each clone class interface informative using different colors and text within a reasonable amount of space in a genealogy. In that way, a developer or a researcher does not have to navigate into a clone class to know about that unless s/he wants to see the code. We also reduce navigation overhead by displaying key information in each view. When designing interfaces for a genealogy, we also display change patterns that may occur during the evolution of code clones in a software system. Then we designed interfaces for investigating clone classes. There are several metrics (e.g., lines of code) that can be presented to learn about a clone class besides the code itself. So, we designed the interface in a way so that a developer or a researcher is able to see enough information about each clone fragment in a clone class without having to look at the code. When a developer wishes to see the code fragments inside a clone class, our interface assists them in seeing the differences between the clone fragments inside that clone class and across revisions as well. Furthermore, we designed our tool to present developer information, and to post and view annotations for particular clone fragments. By building an interactive prototype based on our framework (cf. Chapter 3) we were able to obtain feedback on our approach from clone researchers and experienced developers.

Our contributions described in this chapter, include:

- a new user interface design for multi-touch surfaces to visualize and explore the evolution of code clones in a software system; and,

- a prototype based on our clone genealogy framework and our user interface design in order to obtain feedback from practicing developers and researchers on our approach.

The rest of the paper is organized as follows. Section 4.2 describes the design rationale behind our design choices and major issues. In Section 4.3, we discuss how we built the prototype based on our framework and user interface design. In Section 4.4, we discuss user feedback, when the prototype is useful, and when not. Section 4.5 summarizes our work.

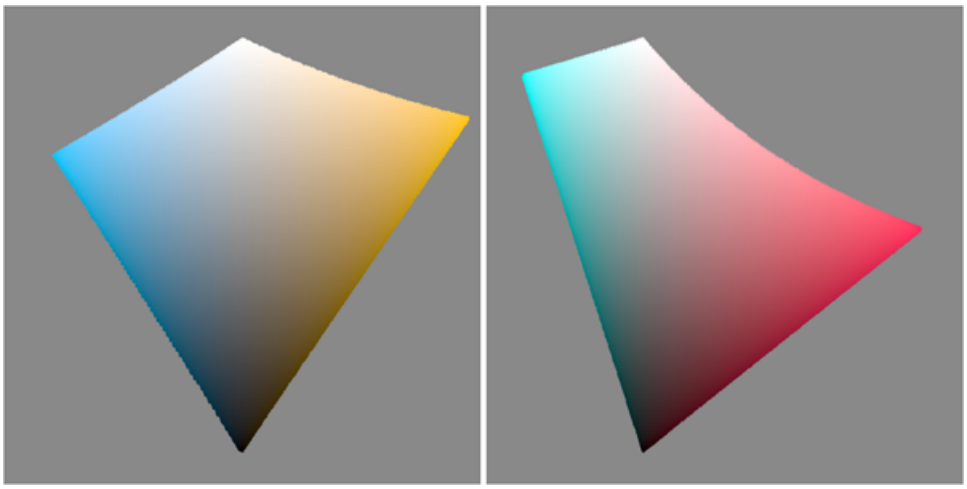


Figure 4.1: Colors perceived identically by people with dichromacy and people with normal color vision

4.2 Design Rationale

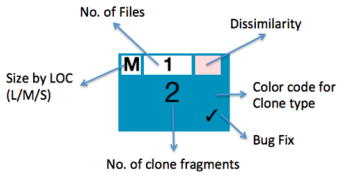
Our main goal is to design an informative user interface so that it makes the study of code clones easier than ever. We previously discussed the basic model for clone visualization in Section 3.3.6 and now we consider several design elements, such as space, color, and text. In this section, we discuss the design rationale behind our design choices and major design issues.

4.2.1 Colors for User Interfaces

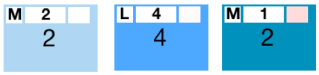
Choosing colors for user interfaces is an important decision as we have a number of situations to consider. First of all, we considered colors that are perceivable by most of the people; otherwise, people with CVD would have difficulty seeing the user interface properly. Statistics say that eight percent of men have reduced sensitivity to the red-green color axis [16]. To resolve this issue, we pick colors from spectral colors (cf. Figure 4.1) that are perceived identically by people with the common forms of CVD, and people with normal color vision [36]. Then, we have to choose the intensity of colors carefully so that a user can easily notice different types of genealogies (e.g., inconsistently changed genealogy). We represent all risk factors with dark colors and safe factors with light colors. For example, we have chosen a light color to represent static clone classes as they are safe and a dark color for inconsistent changes as they are more prone to bugs.

4.2.2 Interface of a Summarized Clone Class

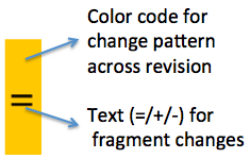
As we discussed earlier in Section 3.3.6, in order to represent a clone class in a genealogy, we summarized each clone class. We think of information of a clone class that can help decide whether we want to explore the code of a clone class or not. We decided to put the type of clone class, number of fragments in a clone class, maximum number of lines of cloned code (LOCC) in a clone class, number of files associated with a clone



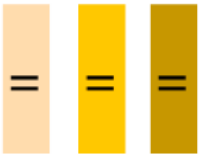
(a) An interface of summarized clone class in a clone genealogy



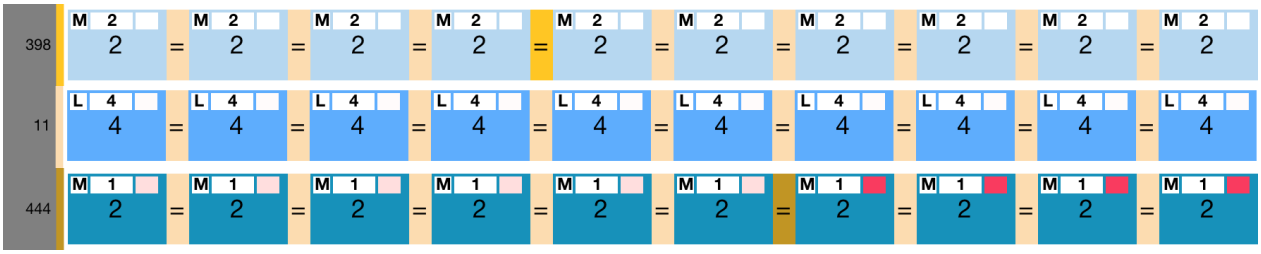
(b) Different types of summarized clone classes: Type-1 (left), Type-2 (middle), and Type-3 (right)



(c) An interface for change patterns across versions in a clone genealogy



(d) Change patterns across: Static (left), Consistent (middle), and Inconsistent (right) versions in a clone genealogy



(e) Clone genealogies: Genealogy of a Type-1 clone class with consistent changes (top), Genealogy of a Type-2 clone class with no changes (middle), and Genealogy of a Type-3 clone class with inconsistent changes

Figure 4.2: Interfaces for a clone genealogy

class, maximum dissimilarities between a pair of clone fragments in a clone class and bug fix information. Figure 4.2a represents an example of a summarized clone class. As we were interested in Type-1, Type-2, and Type-3 clone classes, we present the types of clone classes using colors (cf. Figure 4.2b). We use a light color for Type-1 clone classes as clone fragments in a Type-1 clone class are identical, a dark color for Type-3 clone classes as they need extra care, and we use a color in between of Type-1 and Type-3 clone classes for Type-2 clone classes. We believe that the number of clone fragments and the number of files associated with a clone class would accelerate refactoring decisions. For example, one might be interested in refactoring those clone fragments that are in the same file, in that case, if s/he can see this information prior to viewing the clone class, s/he would be able to decide faster when investigating a large number of genealogies. We considered maximum dissimilarities because that would help find false positive clone fragments easily. We decided to use color instead of text where light color represent high similarity and dark color high dissimilarity. We use dark color for high dissimilarity because it will pop out whenever it comes on the screen so that users can easily identify them. To save space and get rid of large numbers, we categorize clone classes based on LOCC and represent them with letters. Table 3.2 represents categories of clone classes based on LOCC. Furthermore, we were interested to know if a clone class was changed to a fix bug. Thus, we display a *tick* mark on the bottom right corner of a clone class interface if a clone class is changed to fix bug(s). We give each item in a clone class a reasonable size to make sure that everything is properly visible.

4.2.3 Change Patterns

During the evolution of a software system, code clones may change consistently or inconsistently. Representing change patterns in a genealogy is an essential part. To represent how clone classes changed during the evolution of a software system, we take into account consistent changes, inconsistent changes, static clone classes, and fragment changes across versions. We discussed change patterns in Section 2.2.1. Figure 4.2c represents an example of a change pattern interface. If a clone class is static in the next version, then we represent that change pattern interface with a light color, and if a clone class changed inconsistently, then we represent that change pattern with a dark color as inconsistent changes are more prone to bugs. We represent consistently changed clone classes with a color in between light and dark. We consider fragment changes because sometimes we need to know why a new clone fragment is added or why a clone fragment is deleted. If the number of clone fragments in a clone class remains the same in the next version, we put an equal (=) sign on the change pattern interface. Similarly, we put a plus (+) sign if any fragment is added and a minus (-) sign if any fragment is deleted. Figure 4.2d show all types of change patterns in a genealogy.

4.2.4 Interface of a Clone Genealogy

To design an interface for a clone genealogy, we followed the conventional strategy of representing versions of a software system horizontally. We place mapped clone classes horizontally. Then we place change patterns between each pair of mapped clone classes. We give each genealogy a unique number and put them on a bar

on the left so that a user can keep track of genealogies. We also put a little bar beside the genealogy number to inform users whether a genealogy is consistently changed or inconsistently changed. This is because when a genealogy is too long, it will not appear on the screen, and then, this little bar will reduce the effort of scrolling over the genealogy showing how a clone class changed during the evolution. For example, if a genealogy is changed consistently, then the color of that bar will be same as the color of a consistently changed change pattern interface. Figure 4.2e describes how we designed genealogies.

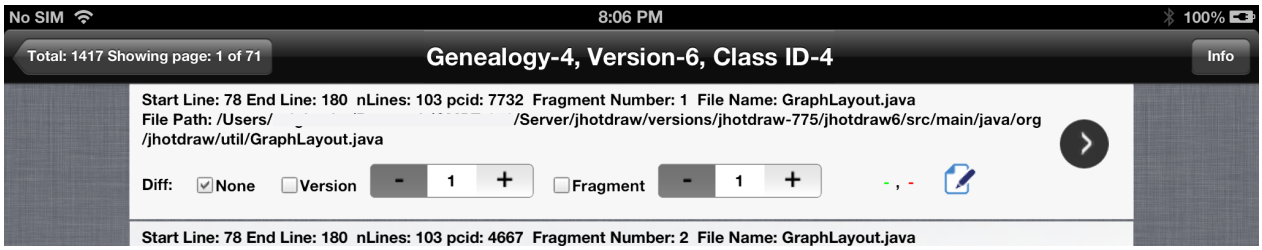
4.2.5 Genealogy Filtering

Filtering clone genealogies of a software system is the fastest way to analyze clone genealogies of interest. We make the filtering interface pretty simple so that users can filter clone genealogies within the genealogy view controller. We take into account types of clone classes, change patterns, and the number of fragments in a clone class for filtering clone genealogies. This interface is basically a table view which contains text with consistent colors (if applicable). For example, the background of Type-1's row is colored with the Type-1 clone class's background color and each row is selectable as well. We include types of clone classes because it is hard to find all genealogies of a specific type from thousands of genealogies. We consider change patterns because, we are interested in knowing how clone classes change during evolution and this filtering option will help us find consistently changed, inconsistently changed, and/or changed genealogies very easily. Finally, we consider the number of fragments because a developer or a researcher may be interested in those genealogies that have too many clone fragments and filtering based on the number clone fragments in a clone class will help find those genealogies.

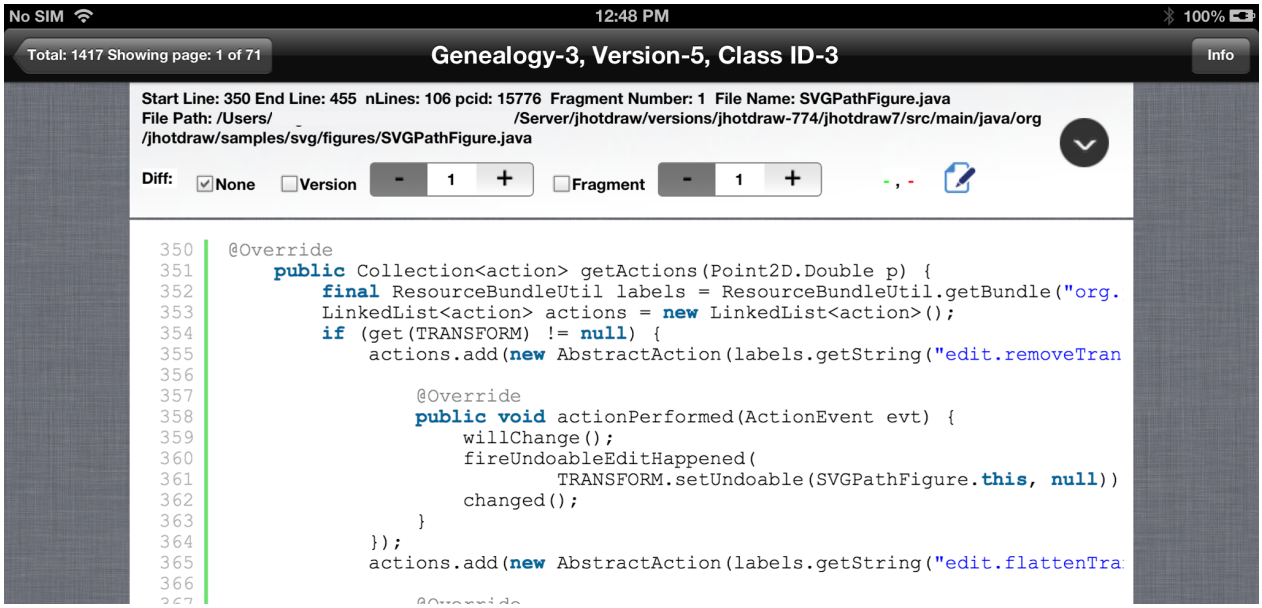
4.2.6 Clone Class Details View

Designing a view to show the code from a clone class was a challenging task, because a clone class contains two or more clone fragments and each clone fragment may contain hundreds of lines. Therefore, we had to think about how can we accommodate all clone fragments efficiently within the space we have. We made an interface (cf. Figure 4.3a) with important information of a clone fragment. We decide to keep the code fragment initially collapsed, but a user can open it whenever s/he wants. In that way, we save space so that a user can see the maximum number of clone fragments at a time. On the other hand, we are also hiding uninteresting code fragments because a user may not want to see all of the clone fragments on a screen. On the interface of a clone fragment, we display a start line and an end line of the clone fragment, number of lines in a clone fragment, file name, and file path. Finally, we place all of the interfaces of the clone fragments vertically in a scroll view so that users can easily scroll up and down on a surface.

Second, we need to know how each clone fragment changed during evolution and how they are different from other clone fragments inside that clone class. We assist users with three options on each clone fragment interface so that s/he can choose what changes s/he wants to see. S/he can choose to see *diff* across versions or with other fragments in that clone class. We use two steppers to see *diff* with another clone fragment;



(a) A clone fragment inside a clone class with code collapsed



(b) A clone fragment inside a clone class with code expanded

Figure 4.3: Inside of a clone class

one is to choose versions and another is to choose fragments. It automatically finds differences on the value change of a stepper across versions or with a fragment within that clone class depending on which option is selected. Therefore, they do not need to get into another clone class to see differences in a genealogy. We display a summary of *diff* results (e.g., number of lines added and deleted) using colored text on the clone fragment interface so that users see *diff* results even if it is collapsed. With an expanded clone fragment interface, they can see more of the line *diff*. Figure 4.4 represents an example of how we visualize the *diff* of two code fragments. Figure 4.4b is the output *diff* result of the two code snippets from Figure 4.4a.

Third, as we view a clone class from a clone genealogy, it would be time consuming if we have to go back to the genealogy interface to view another clone class in the same genealogy. Thus, we have to think about designing interfaces in a way that whenever a user gets into a clone class, s/he can stay there or go over that genealogy without getting back to the genealogy interface. To solve this problem, when a user get into a clone class, we allow them to swipe left and right to visit clone classes from that genealogy. This helps to browse a genealogy while staying in the same interface.

After addressing these issues, we focused on the needs of developers and researchers. While analyzing

```
1  void sum () {
2      int i = 5;
3      int j = 9;
4      int k = i + j;
5      System.out.println("Value: " + k);
6  }
```

```
1  void add () {
2      int a = 5;
3      int b = 9;
4      System.out.println("Value: " + (a + b));
5  }
```

(a) Two code snippet to show *diff*

```
void sumadd () {
    int ia = 5;
    int jb = 9;
    int k = i + j;
    System.out.println("Value: " + k(a + b));
}
```

(b) A *diff* output

Figure 4.4: Visualizing *diff* of two code fragments

clones, a developer or a researcher may need answers to a number of questions. For example, "who changed this version" or s/he may find something interesting about a clone class. In these cases, they may want to contact the developer or they may want to attach some special instructions or notes to some of the clone fragments. With these issues in mind, we added an option on each clone fragment so that anyone can annotate a clone fragment if needed. We also allowed users to see who committed this version and added contact information so that users can send them email directly from the application if needed.

4.3 Building Prototype on A Surface

After designing the user interface, we built a prototype on a surface and elicited feedback on its design and usefulness from developers and researchers. We implemented a client-server architecture where we used a surface as a client so that the prototype can be used from anywhere with an internet connection. In this section, we will describe how we built the prototype.

4.3.1 Choosing a Surface

When selecting a surface, we considered size, availability, cost, touch sensitivity, and stability. In this study, we used an iPad to deploy the prototype. The touch sensitivity of an iPad is remarkably good and it supports fast scrolling over clone genealogies. The iPad we used has a Dual-core A6X with quad-core graphics and 9.7inch (diagonal) LED-backlit MultiTouch display. The configuration of the iPad is good enough for experiencing clone genealogy visualization. Furthermore, because of its portability, we can use it anywhere with internet connection as all the data and the main processing is on the server.

4.3.2 Processes on the Server

We use a server computer on which we construct clone genealogies. To construct clone genealogies, we follow the steps we described in Section 3.3. First, we process versions using settings shown in Table 3.1 for NiCad. Second, we further process all clone classes of a system to categorize clone classes by LOCC, by clone type, to calculate dissimilarity, etc. Third, we map all clone classes between two consecutive versions and automatically identify change patterns. Fourth, we construct genealogies for selected versions of a subject system. Finally, we further process all clone genealogies to retrieve more information for better understanding clone genealogies such as genealogy type, how a genealogy changes, whether a genealogy is a fault fixing genealogy or not, etc. We organized all data according to the model.

4.3.3 Application on iPad

After implementing all the models for constructing clone genealogies, we built an application (the prototype) for iPad. The main challenge of implementing this prototype on iPad was memory. We always had to consider memory issues, since we were working with a huge amount of data. However, we were able to implement the prototype successfully. The application communicates with the server for all services (e.g., get genealogies) as much as it needs. There are several modules of the prototype. They are described as follows:

Menu View Controller

This is the initial view controller of the prototype. This interface allows users to go to either the settings view controller or the genealogy view controller.

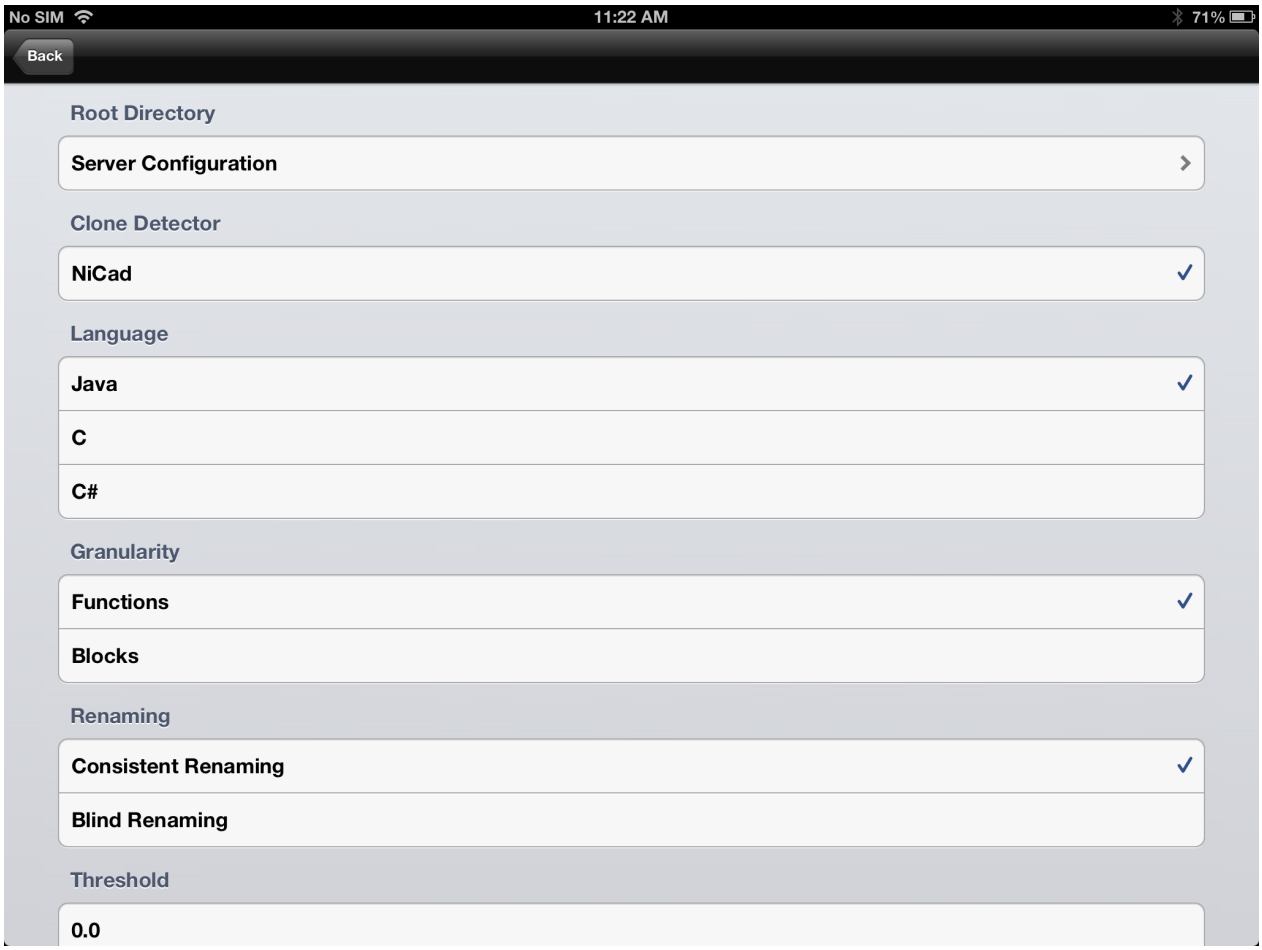
Settings View Controller

We designed this view controller or interface to configure the prototype. To communicate with the server, we need to provide a server name (e.g., IP or domain). It will automatically retrieve the name of all subject systems we have on the server and will allow us to select one of them for analysis. Figure 4.5a depicts how we configure a server and select a subject system for analysis and Figure 4.5b shows some settings of the settings view controller. We give freedom to customize some interfaces in this view controller. Although we do not recommend customizing, we do not want to restrict it either. Users can customize a clone class (e.g., changing color). They can restore the settings to the default at any time. We also provide a help option, so that users can get help if they need.

Genealogy View Controller

This is the view controller where users see clone genealogies of a selected system. As we may have a large number of genealogies, we used paging to visualize genealogies. In this prototype, we load 20 genealogies at a time. We have already described the interfaces related to a clone genealogy (cf., Figure 4.2e) in Section

(a) Server Configuration



(b) Settings in a settings view controller

Figure 4.5: Settings view controller

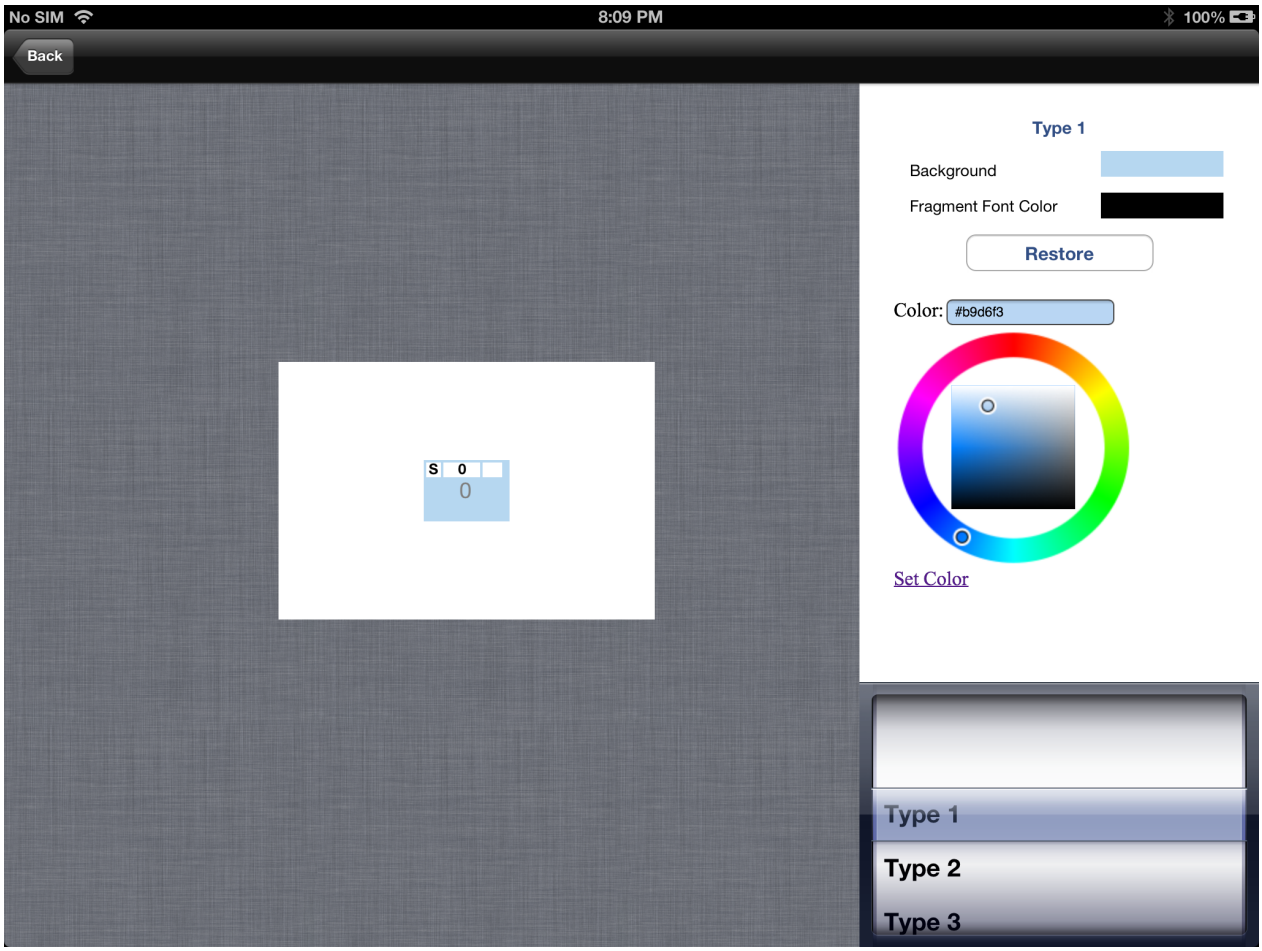


Figure 4.6: Clone Class Customization

4.2.4. We place version numbers on a top bar that moves vertically up and down if a user scroll up and down so that a user never loses track of versions. The bar on the left that displays a unique number for each clone genealogy moves left or right if a user scroll horizontally across versions so that s/he does not lose track of the genealogy at which s/he was looking. At the navigation bar, we display the total number of genealogies and the current page number out of the total number of pages. We also keep a manual scrollbar so that s/he can go to any page at any time. A user can view the details of a clone class (cf. Figure 4.2a) any time with a single tap on that clone class.

Filter View Controller

As we described earlier (in Section 4.2.5), genealogy filtering is an essential for finding clone genealogies of interest. Therefore, we kept this filtering option in the 'Genealogy View Controller' so that users can filter clone genealogies. To save space, we did not make it visible all the time. It pops up whenever a user taps on the 'Filter' button. Then they can select or deselect filtering options. After setting up filtering options, it automatically updates the 'Genealogy View Controller' with the filtered clone genealogies. It will allow

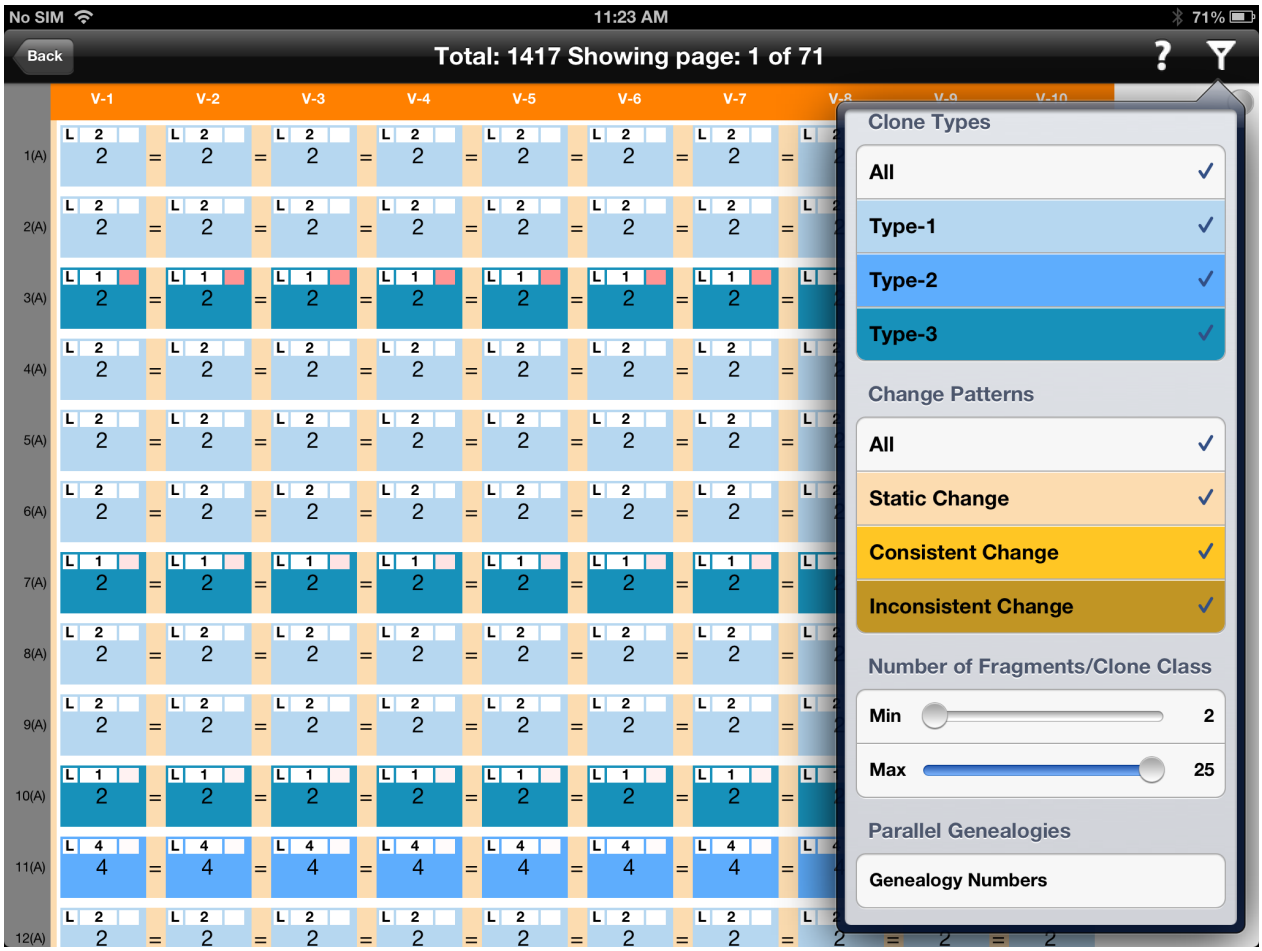


Figure 4.7: Filtering options

users to filter clone genealogies by their type, change patterns, and number of fragments in clone classes. For the ease of filtering by the number of fragments, we put two sliders so that they can quickly select a range. Figure 4.7 shows a genealogy view controller with filtering options.

Clone Class View Controller

When a user taps on a clone class in a genealogy from the 'Genealogy View Controller', s/he initiates the 'Clone Class View Controller'. This view provides useful information regarding a clone class. It contains interfaces (cf. Figure 4.3a) for two or more clone fragments depending on the number of clone fragments. As we described earlier (in Section 4.2.6), each view of a clone fragment is placed vertically on a scroll view so that users can scroll up and down easily. A clone fragment view is initially collapsed so that the user can see the maximum number of clone fragments at a time. A user can see an expanded view (cf. Figure 4.3b) with the source code by tapping on a button. A user can see differences between clone fragments within a class or between the fragments across versions. We have allowed users to select an option whether they want to see across versions or with other fragments. They can always see the summary of *diff* results on the clone

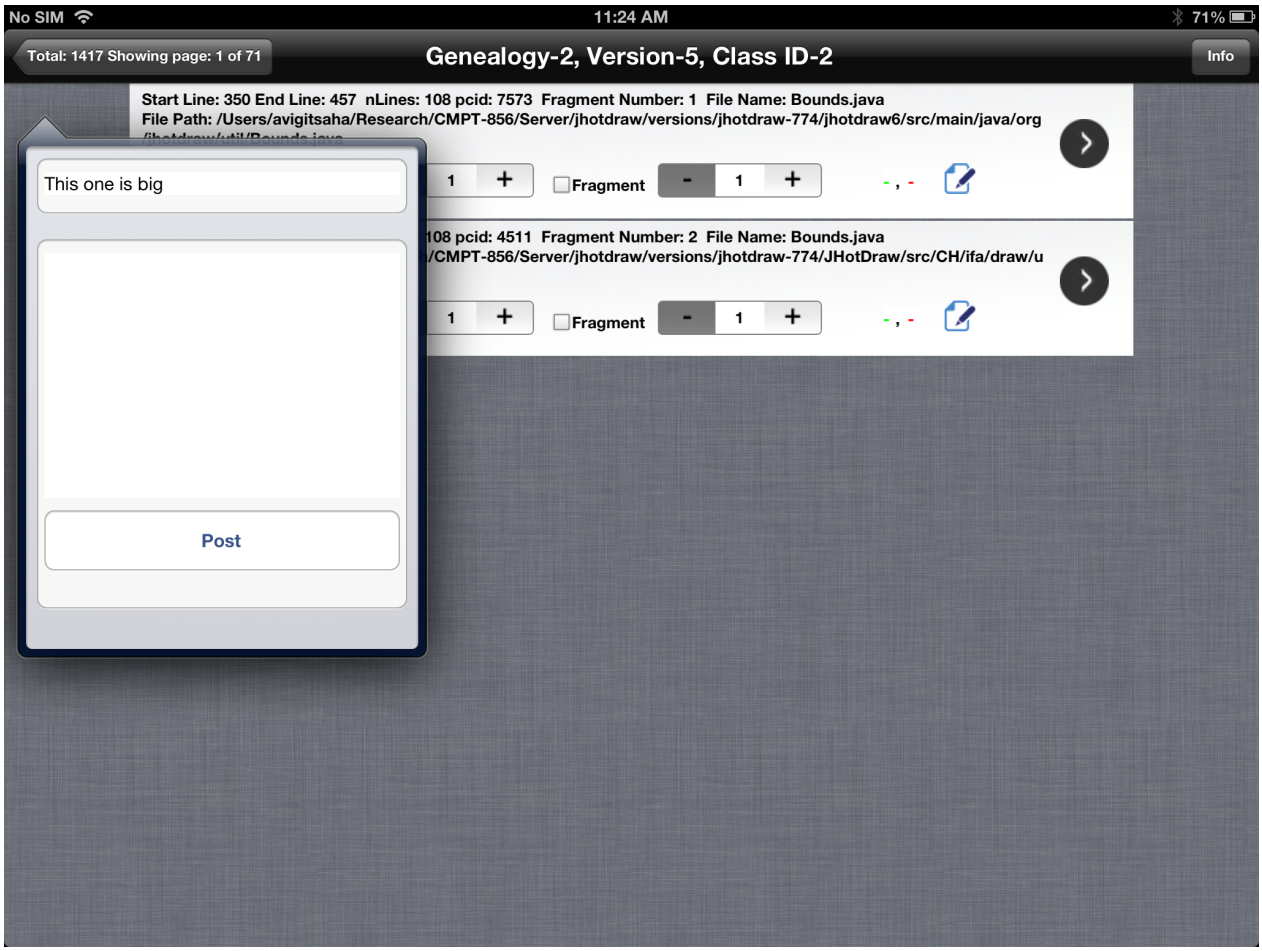


Figure 4.8: An annotation view in a clone class detail view

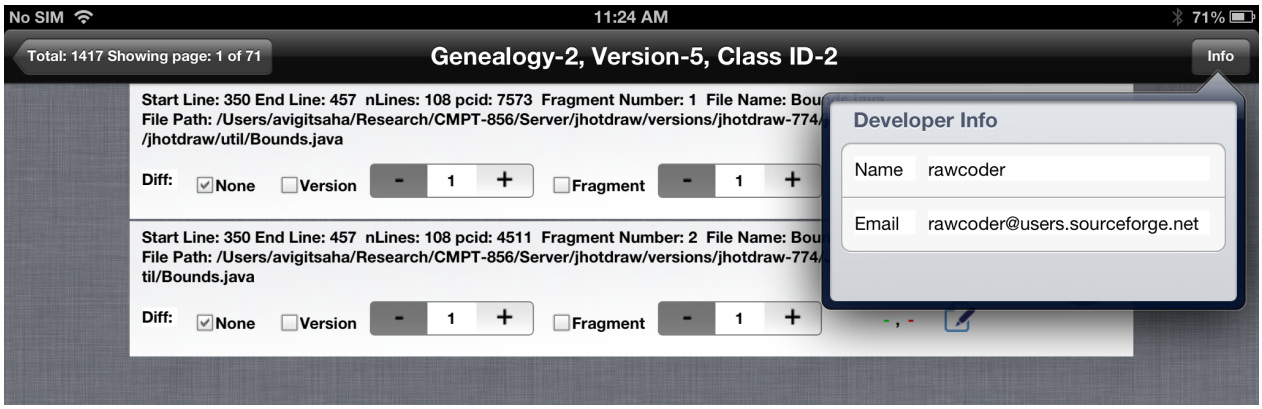
fragment view even if it is collapsed. On the top navigation bar, we also display the genealogy number from which s/he comes from, the current version number, and class id to keep him/her updated.

Annotation View Controller

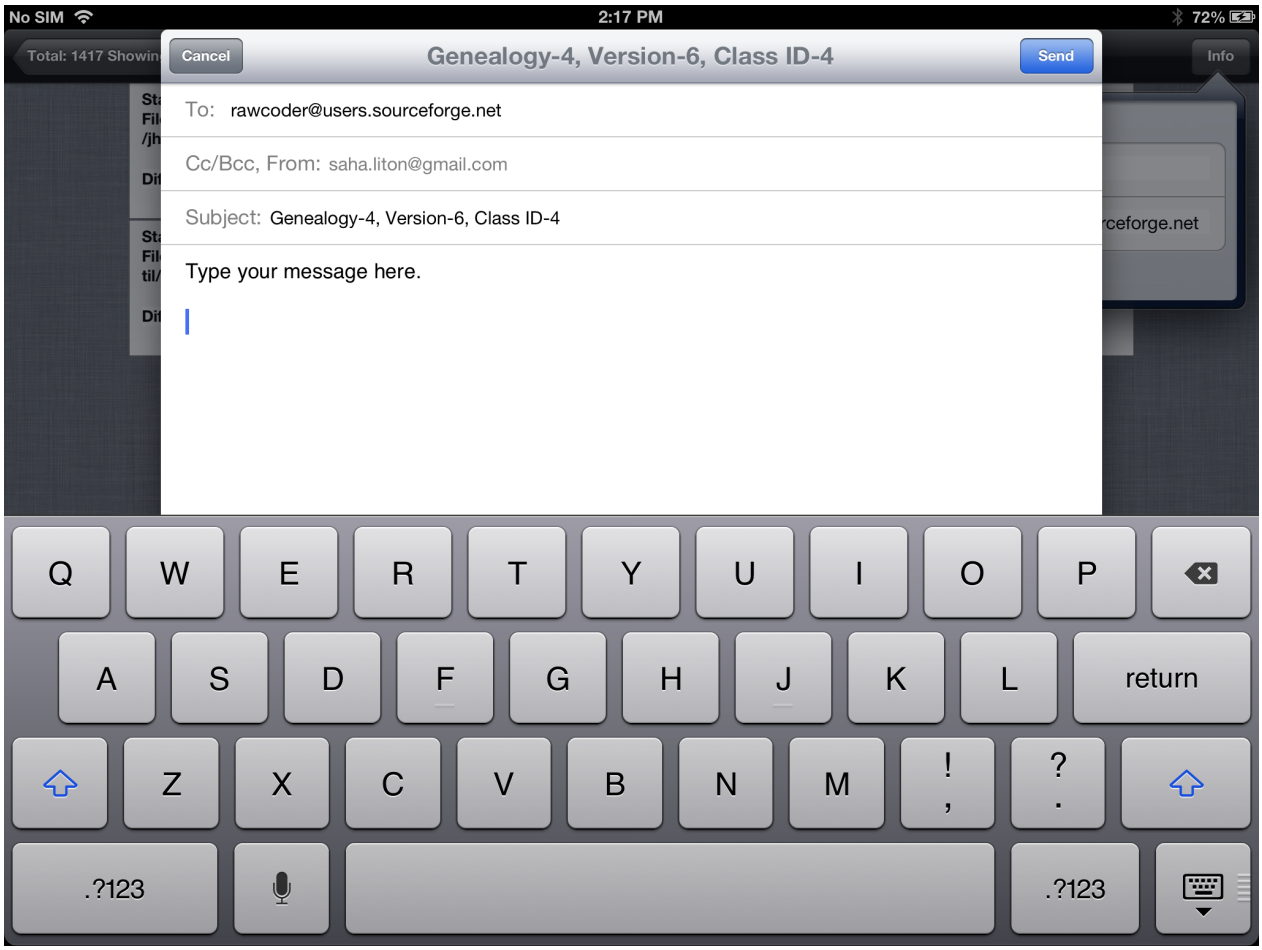
We built the 'Annotation View Controller' to see or post annotations. Whenever a user taps on the annotation button on a clone fragment interface, the 'Annotation View Controller' pops out inside the 'Clone Class View Controller' with annotations if there is any for that clone fragment and a textbox to post new annotations. Figure 4.8 shows an annotation view controller with an existing post and a textbox to write a new post. We store each post in a database so that everybody on a team can see the posts.

Developer's Information View Controller

We built this view controller to show developer information inside the 'Clone Class View Controller'. It appears on a button tap. A user can see who committed this version and the email address of the developer who committed this version. A user can send email to the developer from this view controller if necessary.



(a) A view for showing developer information in a clone class detail view



(b) A view for sending email to the developer in a clone class detail view

Figure 4.9: Developer information

Figure 4.9 shows how we present developer information and how a user can communicate with developers.

4.4 User Feedback

We gathered user feedback to validate designs and the prototype. To gather feedback we designed a structured interview and a semi-structured interview, we conducted 10 structured interviews, and 5 semi-structured interviews.

4.4.1 Structured User Interviews

We conducted 10 structured interviews not allowing one to divert. There were nine graduate students from two different universities, and one faculty member. Nine of the individuals have years of research experience in software engineering. Most of them have research experience on code clones. Six of them have years of industrial experience. We asked them, what information about clone genealogies and clone classes they would find useful. We list the information recommended by the experts to see how much information we provided.

Table 4.1 represents to what extent we could help researchers and developers. We noticed that we considered most of the information the experts recommended, but we also noticed that we missed some information. From Table 4.1, one may argue that we did not provide information as to whether a clone class in a genealogy is refactorable or not, but the information we provided helped users to some extent to decide whether a clone class can be refactored or not. One of the experts mentioned this in an interview. They also asked for the lifetime of a genealogy, which is not displayed in the prototype, but is possible to provide in a genealogy interface. We did not include *late propagation* because it does not occur that often. Furthermore, late propagation occurs due to inconsistent changes, and the prototype is able to visualize inconsistent changes in a clone genealogy. However, the interfaces include some other information (e.g., maximum dissimilarities of a clone class using colors, distribution) about each clone classes in a genealogy that will help them finding genealogies based on numerous attributes.

4.4.2 Semi-Structure User Interview

To understand user's need in more depth, we let them use the prototype and we conducted total 6 hours semi-structured interviews with the researchers, when the prototype would be useful and when not.

When Is The Prototype Useful?

We interviewed 5 researchers and developers. They mentioned situations when they found the prototype useful. A few comments were mentioned often: nice overview of clone genealogies (4 respondents mentioned this); easy to find risky clone classes (1); aids in accelerating refactoring decision (1); liked the remote access (5); quick *diff* (4). In the quotes below, we use numbers as pseudonyms for the interviewed users.

Table 4.1: Comparison with expert's recommendation

About	Information	Expert recommended?	The Prototype Supports?
Clone Class	No. of Clone Fragments	Yes	Yes
	Line of Code	Yes	Yes
	Distribution	Yes	Yes
	Changes Across Versions	Yes	Yes
	File Path	Yes	Yes
	Start Line	Yes	Yes
	End Line	Yes	Yes
	Difference between Fragments	Yes	Yes
	Refacoring Decision	Yes	No
	Changed to Fix Bug?	No	Yes
Genealogy	Change Patterns	Yes	Yes
	Fragment Changes	Yes	Yes
	Late Propagation	Yes	No
	Lineage Information	Yes	No
	Genealogy Type	Yes	Yes
	Type Changes	Yes	Yes
	Life Time	Yes	No
	Bug Relation	Yes	Yes
	Finding problematic clones	Yes	Yes
	Dissimilarity for each Clone Class	No	Yes
	Size by LOC for each Clone Class	No	Yes
	Distribution Summary for each Clone Class	No	Yes

Most of the experts liked the way we represented clone genealogies and found that it would be useful. They thought that we provided enough useful information and one of them mentioned that they found the dissimilarities we showed on each clone class especially useful. One of them mentioned that showing a quick *diff* between fragments across versions and between fragments within a class was useful.

> *I think the tool would be very useful for understanding the evolution of code clones, overall. It first provides a very nice overview about all the genealogies in the code base, which will help me understand the status of the code clones across versions with their change patterns. Then the tool facilitates me to delve deeper into a certain genealogy by providing different useful information such as actual code, different diffs, and so on. (4)*

> *The prototype is useful when the genealogy is viewed over versions. This is because the prototype is providing a birds' eye view with most of the information at one place. (3)*

> *The prototype gives information about a clone genealogy and also gives hints about the changed lines. I can go through different versions of a lineage easily, and I like this. I also found the filtering useful. (2)*

One of them mentioned that the prototype is useful for finding or analyzing risky clone classes, for understanding the distribution of clone classes. They also found the prototype useful for making refactoring decisions.

> *I found this prototype useful for detecting and analyzing the alarming (or risky) clone classes. This prototype also helps me in quick understanding of whether the clone fragments in a particular class are in the same file of in different files. This helps me a lot to make a decision about whether I should refactor the class or not. (1)*

The prototype is useful for remote access. A user can study clone genealogies from anywhere (e.g., in a class room) with an internet connection.

> *To have a quick look at the genealogies with remote access this prototype is very useful. (5)*

When Is The Prototype Not Useful?

After letting them use the tool, we asked 'when is the prototype not useful', so we know what we missed.

The prototype was able to show inline *diff* between two fragments within a clone class and across versions. However, it is less useful if anyone wants to see a side-by-side *diff*. This is also less useful if anyone wants to remove a genealogy that was not changed at all.

> *I want to compare the code side-by-side. Understanding how the line changed was not clear to me except there was no option to remove genealogies that had not changed at all. (2)*

The prototype is not useful for analyzing the exact lifetime of a clone class during the evolution of a software system. Users have to count the lifetime of a clone class manually.

When I want to know about how long the clone class is alive. (3)

We built the prototype on a surface to take advantage of fast scrolling, gestures, and portability. However, one of the experts expected to have a desktop version of the design.

This prototype is possibly not much useful for desktop or laptop users. (1)

The prototype helps finding clone genealogies of interest easily. However, it is less helpful for analyzing multiple genealogies in parallel.

To analyze or review multiple genealogies at a time. (5)

4.5 Summary

In this chapter we discussed a new user interface design and a prototype for visualizing and exploring software clone genealogies based on our framework presented in Chapter 3.

First, our experience shows that most of the tools that help analyze the evolution of code clones, generate a large amount of textual data. It is hard to find genealogies of interest and their change patterns from the large amount of textual data. Thus, we considered how to easily understand the evolution of code clones without spending too much time for processing textual data.

Second, we often need to manually determine how code clones are changed during evolution, especially when looking for inconsistent changes. It is cumbersome, and time consuming to see differences between clone fragments across versions by opening each file or running *diff* manually. That really motivated us to think of an interface that can help us see differences between clone fragments across versions and between clone fragments in a clone class with a single button tap. Then, we designed the clone class interface.

Third, we wanted to get rid of conventional mouse scrolling overhead for horizontal movement, zooming etc. Thus, we chose to build the prototype on a surface because of its extraordinary scrolling capabilities, gesture recognition capabilities, and portability.

Finally, we came up with a new user friendly interface. The new user interface will help us to understand the evolution of code clones. Each interface provides lots of useful information that accelerate decision making. We chose colors for each interface very carefully so that people with the most common forms of CVD, and people with normal vision can see the interfaces properly. After designing the interfaces, we built a prototype using the interfaces and the models we proposed (in Chapter 3) to get feedback from experts. From interviews, we find that the user interfaces are filled with useful information, and they help us understand the evolution of code clones with reduced effort and time. The prototype provides a nice overview of clone genealogies with useful information for each clone class. The prototype would be useful for analyzing inline *diff* between clone fragments across versions and between clone fragments within a clone class.

Chapter 5

An Empirical Investigation into the Evolution of Function Clones

In the previous chapter, we represent how we build a prototype for a multi-touch surface using our framework and user interfaces to visualize clone genealogies in a software system. We also show that our prototype is useful for finding interesting clone genealogies. In this chapter, we use our framework and prototype to investigate how function clones evolve during the evolution of a software system. In this chapter, we discuss our findings and how the prototype helps us to find interesting patterns.

5.1 Motivation

Since our framework and prototype are useful for finding patterns from the evolution of code clones, we use them to conduct this empirical investigation to understand the evolution of function clones in software systems in order to validate their effectiveness. Understanding the evolution of code clones is important to manage clones properly. There are several studies in this regard. Most of these studies investigate how clones evolve during the evolution of a software system by constructing clone genealogies. These studies help us understand and maintain clones in a number of ways such as understanding the changing behaviour of clones and developing new tools to manage clones. The more patterns we can discover in clone genealogies, the better we will be able to manage clones efficiently and effectively. However, most of the existing studies are limited to Type-1 and Type-2 clones [8], [70], [75], [108], [116]. Recently Saha et al. [110] conducted an empirical study to understand the evolution of Type-3 clones in software systems. Clones can also be considered at different levels of granularity, such as function clones or block clones. In our study we will focus on only function clones.

In order to investigate the evolution of function clones more rigorously, we distinguish four types of functions. There can be four types of functions based on their return type and parameters. A function could have no return type and no parameters, no return type and some parameters, a return type and no parameters, and return type and some parameters. We use these function types to categorize function clones. The classification of function clones will be discussed in Section 5.2. After categorizing function clones, we construct clone genealogies across releases of a software system. Then we investigate those clone genealogies to find patterns that can help maintain function clones. We investigate how function clones evolve during

the evolution, and see if we need to care about any of the categories of function clones. We represent the findings by answering three research questions as follows:

1. *Which categories of function clones do developers create most often and how long-lived are they?* By answering this question we hope to determine if developers have a tendency to create certain categories of function clones because this information may help prevent us from creating new function clones. We also see how long-lived they are so that we can manage them properly.

2. *Which categories of the function clones are most important to look at?* By answering this question, we want to see which categories of function clone genealogies exist more than other types of genealogies and how they change over time so that we know if we should pay extra attention to any clone genealogies while maintaining code clones in a software system.

3. *How consistently do long lived function clone genealogies change during their evolution?* By answering this question we can see whether most of the long lived clone genealogies changed consistently or not because we know that the genealogies that are long lived and are changed consistently are not easily refactorable [70].

4. *Do function clones convert to other function clone categories?* From the results of the previous two questions, we are motivated to find an answer to this question. We observed function clone genealogies to see if they converted to other function clone genealogies at some point in their evolution.

The rest of this chapter is organized as follows. In Section 5.2, we classify function clone classes into five categories and formally define them with examples. In Section 5.3, we discuss an approach for answering the research questions. Section 5.4 describes the analysis of the evolution of function clones and discusses implications of the results by answering the research questions. Section 5.5 describes how we utilize our framework and prototype to conduct this study. Section 5.6 describes limitations of this study. Section 5.7 concludes this study.

5.2 Classification of Function Clones

There are four types of clone classes based on the degree of textual, syntactic, and semantic similarity among clone fragments. They are Type-1 (exact), Type-2 (Type-1 with renamed identifiers), Type-3 (Type-2 with added or deleted lines) and Type-4 (semantically similar). In this study, we further classified function clones into the following five categories based on their function types.

- **FCType-1:** a clone class that contains function clones with no return type and no parameters.

- **FCType-2:** a clone class that contains function clones with no return type and one or more parameters.

- **FCType-3:** a clone class that contains function clones with a return type and no parameters.

Table 5.1: Examples of Function Clone Classes

Types	Clone Class		
	Fragment 1	Fragment 2	Fragment 3

FCType-1	`void foo () {`
	`  int a = 0;`
	`  for(int i=0;i<10;i++){`
	`  a=a+i;`
	`  }`
	`  return;`
	`}`
		`void foo () {`
		`  int a=0;`
		`  for(int i=0;i<10;i++){`
		`  a=a+i;`
		`  }`
		`  return;`
		`}`
			`void foo () {`
			`  int a=0;//initialize`
			`  for(int i=0;i<10;i++){`
			`  a=a+i;`
			`  }`
			`  return;`
			`}`

FCType-2	`void foo (int n) {`
	`  int a=0;`
	`  for(int i=0;i<n;i++){`
	`  a=a+i;`
	`  }`
	`  return;`
	`}`
		`void foo1 (int n) {`
		`  int a=0;`
		`  for(int i=0;i<n;i++){`
		`  a=a+i;`
		`  }`
		`  return;`
		`}`
			`void foo2 (int n) {`
			`  int a=0;//initialize`
			`  for(int i=0;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return;`
			`}`

FCType-3	`int foo () {`
	`  int a=0;`
	`  for(int i=0;i<10;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo1 () {`
		`  int a=0;`
		`  for(int i=0;i<10;i++){`
		`  a=a+i;`
		`  }`
		`  a=a*10;`
		`  return a;`
		`}`
			`int foo2 (int n) {`
			`  int a=0;//initialize`
			`  for(int i=0;i<n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

FCType-4	`int foo (int n) {`
	`  int a=0;`
	`  for(int i=1;i<=n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`int foo (int n) {`
		`  int a=0;`
		`  for(int i=1;i<=n;i++){`
		`  a=a+i;`
		`  }`
		`  return a;`
		`}`
			`int foo (int n) {`
			`  int a=0;`
			`  for(int i=1;i<=n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

FCType-5	`int foo (int n) {`
	`  int a=0;`
	`  for(int i=1;i<=n;i++){`
	`  a=a+i;`
	`  }`
	`  return a;`
	`}`
		`void foo (int n) {`
		`  int a=0;`
		`  for(int i=1;i<=n;i++){`
		`  a=a+i;`
		`  }`
		`  return;`
		`}`
			`int foo (int n) {`
			`  int a=0;`
			`  for(int i=1;i<=n;i++){`
			`  a=a+i;`
			`  }`
			`  return a;`
			`}`

- **FCType-4:** a clone class that contains function clones with a return type and one or more parameters.

- **FCType-5:** a clone class that contains a mix of function clone types.

5.3 Experimental Setup

5.3.1 Subject Systems

We select three popular open source software systems for this study. All of the subject systems have a long development history and have been used in several studies. In this section, we will briefly describe those subject systems.

Table 5.2: Subject Systems

System	No. of Releases	Start Release	End Release	LOC
Ant	19	1.5.2	1.9.1	74046 - 130323
ArgoUML	9	0.33.1	0.34	193302 - 195363
JHotDraw	10	5.4	7.5.1	28103 - 137837

Ant[1] is a Java library and command-line tool that drives processes described in build files as targets and extension points dependent upon each other. The main use of Ant is to build Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. It has over 130323 LOC.

ArgoUML[2] is the leading open source UML modelling tool and includes support for all standard UML 1.4 diagrams. It runs on any Java platform and is available in ten languages. It has over 195K LOC.

JHotDraw[3] is an open source software system written in Java. It is a GUI framework for technical and structured graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. It has over 137837 LOC.

All of the subject systems have over 100K lines of code covering different domains to avoid biased results. We conducted this study at the release level because the source code is expected to be in a stable form and thus any inconsistent changes to clone fragments between two releases should be either intentional or accidental. Therefore, we have chosen release level instead of revision level for this study. Table 5.2 shows details of the subject systems.

5.3.2 Clone Detection

As we have discussed in Section 3.3.1, to detect clones from all releases we used NiCad [22] because it has already been shown to be effective in detecting near-miss clones while maintaining high precision and recall [103], [104], [101]. We carefully chose parameters for NiCad as described in Table 3.1. We set granularity to functions, minimum clone length to 5 LOC, dissimilarity threshold to 30% and we also applied consistent renaming. Before detecting clones, we process all releases of the subject systems to remove all test files so that we do not get false positive clones in this experiment.

5.3.3 Extraction of Clone Genealogies

After detecting clones from subject systems, we construct clone genealogies for further investigation. We follow the processes we described in Section 3.3. First, we process all XML outputs generated by NiCad.

[1] **http://ant.apache.org/**
[2] **http://argouml.tigris.org/**
[3] **http://www.jhotdraw.org/**

Then, we classify all the clones of a subject system based on function types as we described in Section 5.2. To classify a clone class, we take each function clone and extract the return type and parameters of the function. If all of the functions of a clone class are not the same type, we mark the clone class as FCType-5. Otherwise, we mark the clone class according to the function type as described in Section 5.2. For example if clone class has all function clones with no return type and no parameters, we mark that clone class as an FCType-1 clone class. After classifying all clone classes, we map clone classes between consecutive releases. To map clone classes, we map all functions between two consecutive releases, then using the function mapping data, we map all clone classes between two consecutive releases. During this process, we automatically identify change patterns of a clone class such as consistent changes and inconsistent changes. The process of mapping clone classes and identifying change patterns was already discussed in Section 3.3.3. After mapping clone classes, we construct genealogies using them (cf., Section 3.3.4). Then, we identify the genealogy type and overall change pattern of each genealogy.

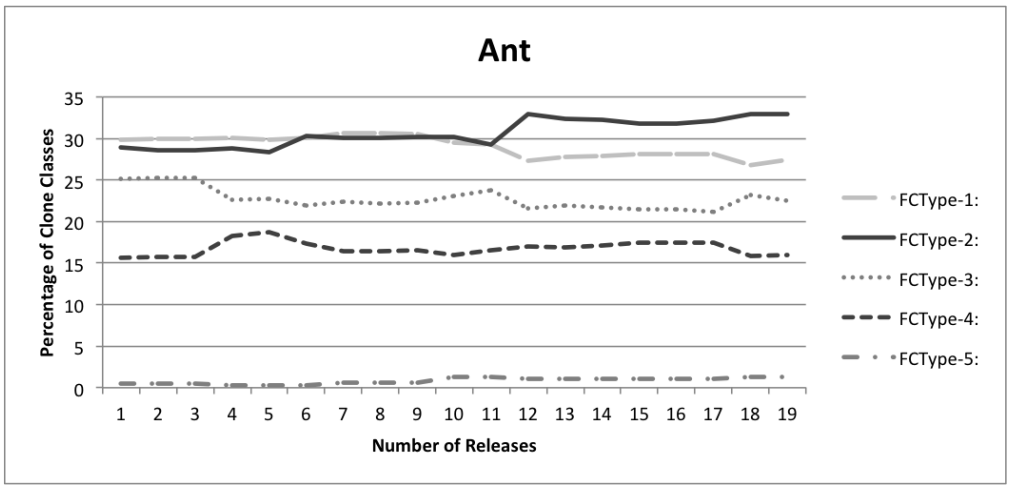
5.4 Results

In this section, we will discuss answers to the research questions in detail.

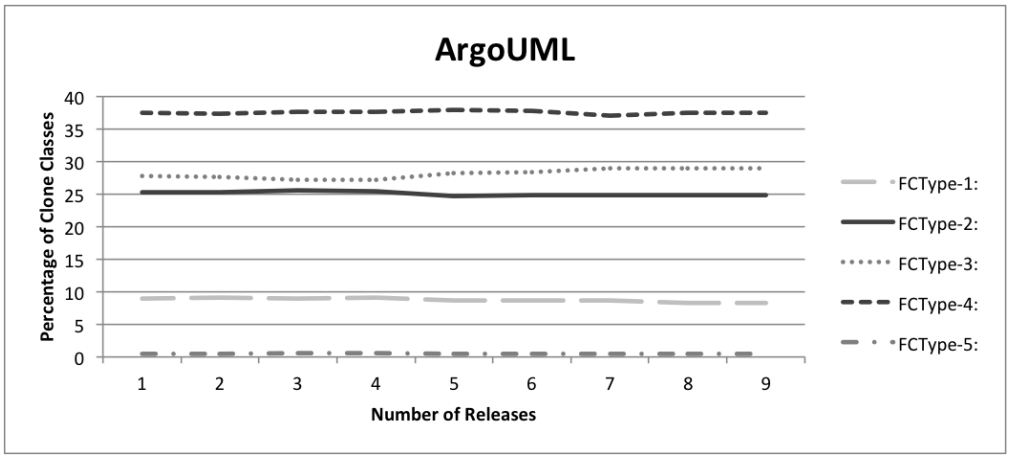
5.4.1 RQ1: Which categories of function clones do developers create most often and how long-lived are they?

We calculate the percentages of each category of function clones across releases. We plot collected data on a graph where the x-axis represents the sequential number of releases and the y-axis represents the total number of clone classes. We repeat this process for each subject system. Then, we find the percentage of long lived clone genealogies for each category of function clone and represent using a bar chart. Finally, we analyze the data to answer this question.

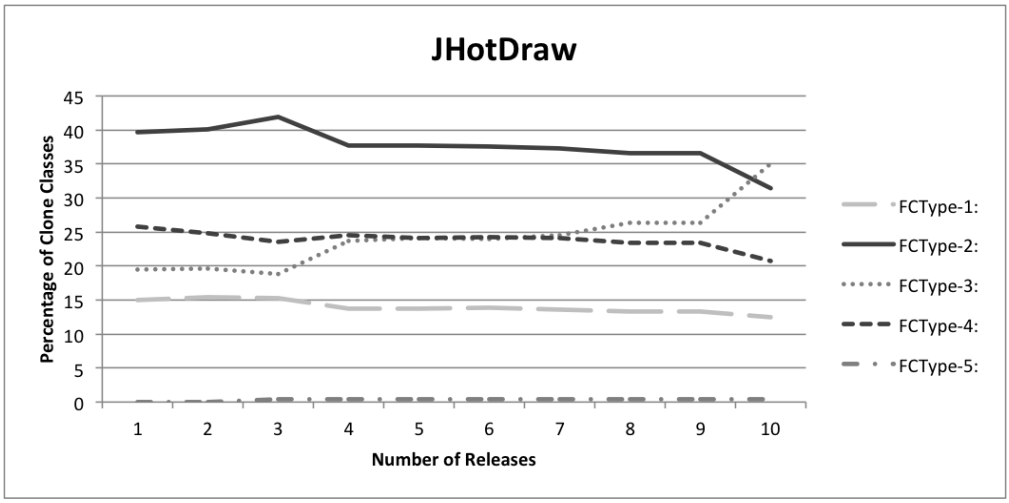
A study shows that overall clone density increases over time [78]; therefore, we are interested to know which categories of function clones developers mostly create over time. We collect data for each release, then plot all data on a graph to represent the result. Figure 5.1 represents results for all subject system. From the Figure 5.1, we can see that the FCType-2 grows fast over time. Furthermore, Figure 5.1a and Figure 5.1c show that the percentages of FCType-2 in Ant and JHotDraw are higher than that of other categories of function clones. We can also see from Figure 5.1 that the percentages of FCType-4 over time is significant but not as significant as FCType-2. From the result, we can say that developers have more tendency to create clones of FCType-2 than that of FCType-4. However, there are a few FCType-5 clones, which means there are few clone classes that contain different types of functions. As we have seen developers create FCType-2 and FCType-4 mostly. We also investigate their lifetime to see how long they live. Figure 5.2 depicts our result. We see that the percentages of FCType-2 ranges from 53% to 93% and percentages of FCType-4 varies from 51% to 82% in the subject systems. We also see that most of the subject systems have more than



(a) Growth of all types of function clones of *Ant*



(b) Growth of all types of function clones of *ArgoUML*



(c) Growth of all types of function clones of *JHotDraw*

Figure 5.1: Growth of function clones

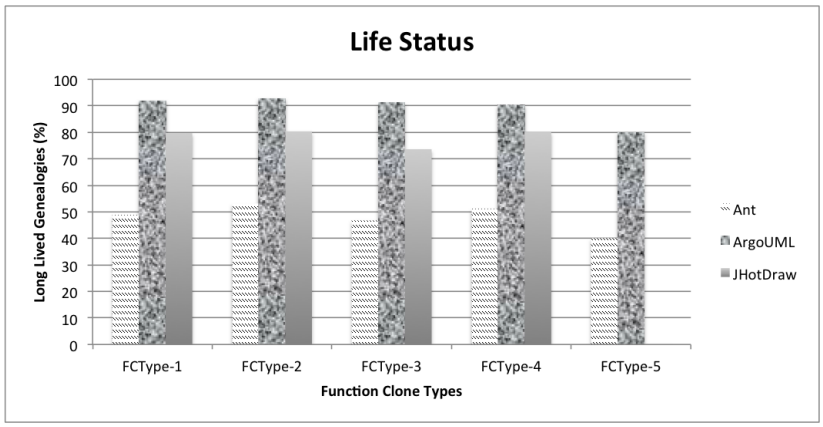


Figure 5.2: Percentage of long live clone genealogy for each subject system

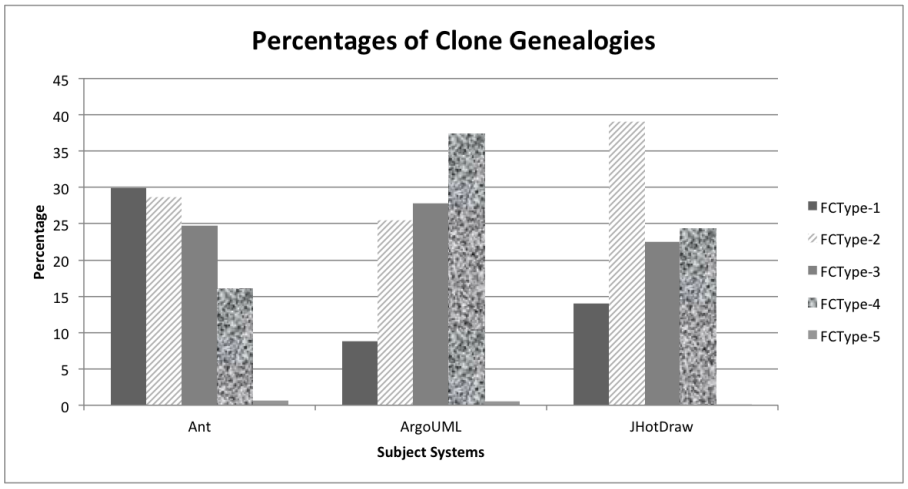


Figure 5.3: Percentage of different types of clone genealogies across releases of different software systems

70% long lived genealogies.

5.4.2 RQ2: Which categories of the function clones are most important to look at?

We calculate the percentage of each category of clone genealogies while constructing clone genealogies for each subject system. Then, we represent the results using a bar chart to see how the percentage varies among different subject systems and whether any categories of clone genealogies need extra attention while maintaining the code clones in a software system.

During construction of clone genealogies of a subject system, we identify the genealogy types and count each category of genealogy. Figure 5.3 represents the percentages of different types of clone genealogies across releases of the software systems. From Figure 5.3, we find that there are FCType-2 clone genealogies ranging from 29% to as high as 39%. As we have already seen that most of the FCType-2 genealogies are long

Table 5.3: Change patterns of the function clones (SG = Static Genealogy, CCG = Consistently Changed Genealogy, and ICG = Inconsistently Changed Genealogy)

Software System	Category	SG (%)	CCG (%)	ICG (%)
Ant	FCType-1	52.61	9.3	38.26
	FCType-2	49.54	8.64	41.81
	FCType-3	51.57	11.05	37.36
	FCType-4	51.21	8.13	40.65
	FCType-5	40	20	40
ArgoUML	FCType-1	87.35	2.29	10.34
	FCType-2	86.85	1.19	11.95
	FCType-3	85.81	2.91	11.27
	FCType-4	85.90	1.89	12.19
	FCType-5	100	0	0
JHotDraw	FCType-1	64.05	14.46	21.48
	FCType-2	61.21	15.01	23.77
	FCType-3	64.59	14.98	20.41
	FCType-4	59.90	17.42	22.67
	FCType-5	100	0	0

lived, we can say that FCType-2 clones need extra care while managing code clones. On the other hand, we see that there is a negligible amount of FCType-5 clone genealogies. We also see from the Figure 5.3 that there is about 22% to 28% FCType-3 clone genealogies in the subject systems. Unlike FCType-3, FCType-4 clone genealogies shows a scattered pattern in the subject systems. The percentage of FCType-4 clone genealogies varies from 16% to 38%. In order to know how they change over time, we investigate their change patterns. From Table 5.3, we can see that there are more inconsistently changed genealogies than consistently changed genealogies. However, we also see that most of the genealogies are static for all categories. Since the percentages of FCType-2 and FCType-4 is high, and the percentage of the inconsistently changed FCType-2 and FCType-4 clone genealogies are also high, we can say that they are less stable than the other categories of function clones. Therefore, they need extra attention while managing code clones in software systems as they seem to be vulnerable.

5.4.3 RQ3: How consistently do long lived function clone genealogies change during their evolution?

To answer this question, we investigate all the long lived clone genealogies and their change patterns. We represent our findings in Table 5.4. After investigating the long lived genealogies in the subject systems we find 1.28% to 21.72% of the total long lived clone genealogies changed consistently. We also see that more

Table 5.4: Change patterns of the long lived function clones

Software System	Category	Total Genealogies	CCG (%)
Ant	FCType-1	112	14.28
	FCType-2	117	14.52
	FCType-3	89	17.97
	FCType-4	63	11.11
	FCType-5	2	50
ArgoUML	FCType-1	80	17.61
	FCType-2	233	17.92
	FCType-3	251	18.94
	FCType-4	333	21.72
	FCType-5	4	0
JHotDraw	FCType-1	193	2.5
	FCType-2	541	1.28
	FCType-3	285	3.18
	FCType-4	336	2.10
	FCType-5	0	0

than 11% of the total long lived genealogies changed consistently in Ant and ArgoUML. As we know that the consistently changed long lived clone genealogies are not easily refactorable [70], we conclude that we can try to refactor most of the long lived function clones since most of the long lived clone genealogies did not change consistently, especially the FCType-2 and the FCType-4 clones as they are less stable than other categories in the software systems.

5.4.4 RQ4: Do function clones convert to other function clone categories?

In this study, we are interested to know if most of the function clones convert to other categories of function clones during evolution. During construction of clone genealogies, we track each genealogy to see if they change their type over time. Finally, we calculate the percentage of changes and represent the data with a bar chart to see the result.

Since we have seen that the growth of FCType-2 clones is higher compared to the other clone classes, and there are FCType-2 clone genealogies ranging from 29% to as high as 39% of the total genealogies, we are interested to see if there are any categories of clone classes that are changed to FCType-2 during their evolution. Table 5.5 represents the results. We can see that in ArgoUML 74.59% of the total changed genealogies were converted to the FCType-2 clone genealogies. In Ant, 71.34% of total changed genealogies were converted to the FCType-2 clone genealogies and in JHotDraw, the percentage was 60.15%. Finally, we can conclude that a large portion of clone genealogies converts to FCType-2 genealogies at some point due

Table 5.5: Genealogy Conversions

Subject System	Total Converted Genealogies	Converted to FCType-2	Converted to FCType-2 (%)
Ant	677	483	71.34%
ArgoUML	921	687	74.59%
JHotDraw	1468	883	60.15%

to changes over time and this could be a reason for which FCType-2 clone classes increased across releases of the software systems over time.

5.5 Contribution of the Framework and Prototype

We extended our framework to work with the new categories of function clones. Thus, our framework is equally useful for finding patterns from the evolution of different categories of function clones as it is useful for finding patterns of Type-1, Type-2, and Type-3 clones. We use the prototype for initial investigation so that we can find patterns easily with the help of the filtering options. We investigate change patterns, lifetime, number of genealogies, etc. Based on our findings we further analyze the data and present the outcomes in this chapter. Since our framework provides JSON output, it was easy for us to analyze the data.

5.6 Study Limitations

In this section, we discuss limitations of this study.

5.6.1 Clone Detection

The results of this study depend on raw clone data generated by NiCad. We have chosen NiCad because it has already been effective to detect near-miss clones while maintaining high precision and recall [103], [104], [101]. However, certainly, NiCad may miss some clones and as a result they will also be ignored in this study.

5.6.2 Mapping Clone Classes

To construct clone genealogies, we map clone classes, and we implemented the same approach proposed by Saha et al. [109]. In that case, a clone class may be mapped wrongly because of an ambiguous situation. However, we performed an extensive manual investigation to validate mappings of clone classes and found that these situations are very rare.

5.6.3 Subject Systems

In this experiment, we only studied Java open source software systems. Therefore, the findings may not apply to other software systems that are written in other programming languages.

5.7 Summary

In this study, we investigated the behaviour of function clones across releases using three open source software systems. Although, studies of code clone evolution are not new where most of the studies of code clone evolution considered Type-1 and Type-2 clones, and some of them considered Type-3 clones as well; however, in this study, we further classify function clones, and investigate their behaviour across releases using three open source software systems.

First, we classify clone classes into five different categories based on function type such as if a clone class contains only function clones with no return type and no parameters, we categorize that clone class as FCType-1 clone class. After that, we investigate the behaviour of each category of clone class over time by answering three research questions.

We analyze all function clones to understand what categories of function clones developers mostly create. This analysis shows that developers have the tendency to create FCType-2 clone classes. We find that the percentages of FCType-2 clones increases over time compared to other function clones. However, developers also create a significant number of FCType-4 clones. We also find that there is about 53% to 93% of long lived FCType-2 genealogies and 51% to 82% of FCType-4 long lived genealogies in the subject systems. We further investigate what categories of clones need extra care while managing code clones. We find that there are FCType-2 clone genealogies ranging from 29% to as high as 39%, and FCType-4 clone genealogies ranging from 16% to 38%, and furthermore there are more FCType-2 and FCType-4 inconsistently changed clone genealogies than FCType-2 and FCType-4 consistently changed genealogies. We conclude that they need extra attention while maintaining code clones in a software system. We are interested in knowing if the long lived genealogies are changed consistently or not. Our results show that only 1.28% - 21.72% of the total long lived genealogies changed consistently that implies developers can try to refactor most of the long lived clone genealogies. We are also interested to know if any clone genealogies changed to other categories of clone genealogies during their evolution and we find that about 60% to 75% of converted clone genealogies were converted to FCType-2, and this could be another reason for the higher number of the FCType-2 clones.

By conducting this study, we see that the framework and prototype are useful for finding patterns easily and efficiently. Furthermore, the JSON output of the framework makes it easy to further analyze the patterns.

Chapter 6

Bugs Due to Clones

In the previous chapter, we conducted an empirical investigation into the evolution of function clones using our framework and prototype. Since, our framework and prototype can help us to find buggy clone genealogies easily, we use the framework and prototype to conduct a second empirical study to see how clones are related to bugs. In this chapter, we discuss our findings by answering four research question and how the prototype helps us in this study

6.1 Motivation

In the previous chapter, we categorized function clones and analyzed their evolution in software systems. However, in this chapter, we investigate how function clones contribute to bugs. Clones are often created by copying and pasting practices of programmers. In previous studies, it was found that clones not only cause extra effort in maintenance activities [68], but also they are considered to be a 'bad smell' [10], [60], [37]. For example, if a code fragment is buggy, all other fragments copied from it may replicate the same bug. Therefore, it is important to make sure that developers modify all clones of a buggy clone class properly while fixing a bug. However, it is difficult to remember all clones if there are many of them in a software system.

From previous studies, we find that software systems can have duplicated source code in amounts ranging from 5-15% of the code base [105] to as high as 50% [99]. Although, clones can be removed by automatic refactoring [72], [48], [11]; however, long lived consistently changing clones are not easily refactorable [70]. Furthermore, non buggy clone code could be buggy after cloning [53]. Despite the fact that cloning has disadvantages, developers are cloning code because of other advantages. Therefore, we need to study the history of a clone class as well as a buggy clone class to mitigate the risks of cloning.

In this study, we are interested in buggy clone classes in three subject systems to assess clones impact on defect occurrence of software products. We investigate how clone classes contributed to bugs and how developers modified buggy clone classes to fix bugs. We also try to find a relationship between the cloning rate for the buggy clone classes and the non-buggy clone classes. Furthermore, we categorize buggy clone classes based on the number of clone fragments to see if there is any category that is more in numbers than other categories. We represent our findings by addressing the following four research questions.

1. ***RQ1: To what extent are clone classes related to bugs?***

   We investigate the subject systems to see whether the majority of clone classes are related to bugs or not. In a recent study, Rahman et al. [98] claimed that on average more than 80% of bugs contained no cloned code. However, they did not investigate the percentage of cloned code that contributed to bugs. For example, if all of the clone classes contribute to bugs and the amount of bugs created by the clones is only 2% of the total bugs in the system, then the clones will be considered as bad smell because 100% of the total clones are contributing to bugs. Therefore, it is important to know to what extent clones are related to bugs. We show that there is less than a 40% chance that there will be no buggy clones.

2. ***RQ2: How are buggy clones managed?***

   Thummalapenta et al. [116] showed that developers are actually quite aware about updating clones consistently, even if they reside in different files of the software system. On the other hand, Saha et al. [108] showed that only 11-38% of clone genealogies were changed consistently. Therefore, we investigate how buggy clone classes changed to fix a bug because it is important to ensure that all the necessary changes are propagated to all of the clone fragments of a buggy clone class. We show that most inconsistent changes are made unintentionally and most of the time the clone fragments are in separate files.

3. ***RQ3: Is there any relationship between the growth of buggy clone classes and the growth of non-buggy clone classes over time?***

   We try to find out if the number of buggy clone classes increase or decrease proportionally with the non-buggy clone classes in the software systems. A previous study [81] showed that clones increase over time and if buggy clone classes also increase over time, then it would be difficult to maintain the software system. Our result shows that they are not related to each other.

4. ***RQ4: Which category of buggy clone classes are more buggy than others?***

   Rahman et al. [98] reported that they found no evidence that support the claim that prolific clones (clones with many copies) have more buggy code than the non-prolific ones (clones with fewer copies). First of all, they did not clearly state the ranges of the number of clone fragments for prolific and non-prolific, which makes their result vague. Therefore, we categorize of the buggy clone classes into 'Large', 'Medium' and 'Small' based on the number of clone fragments to analyze what amounts of clone fragments generally buggy clone classes contain so that developers can be careful while modifying the categories of buggy clone classes that are more prone to bugs than other categories. We show that the 'Small' and the 'Medium' categories of buggy clone classes are more prone to bugs than the 'Large' category.

The rest of this chapter is structured as follows: Section 6.2 describes our experimental setup. Section

Table 6.1: Subject Systems (Fault Fixing Revision = FFR)

Subject	# of FFR	Start Revision	End Revision	Start Date	End Date	Duration	SLOC
Ant	1001	274319	1295749	March 27, 2003	March 03, 2012	9 years	128208
dnsjava	215	1	1665	September 06, 1999	November 15, 2011	12 years	23738
JHotDraw	80	42	779	April 30, 2002	April 16, 2012	10 years	139870

6.3 describes our results. Section 6.4 describes how we utilize our framework and prototype to conduct this study. Section 6.5 describes threats to the validity of our work and finally, Section 6.6 concludes our work.

6.2 Experimental Setup

In this section, we discuss experimental setup to collect data to answer our research questions. It includes choice of our subject systems, settings of clone detector, procedures of data collection and investigation.

6.2.1 Subject Systems

We studied three open sources software system. To select subject systems we gave preferences to those subject systems that have a long development history, are different in size and are different in domain. Based on our preferences we choose Ant, dnsjava, and JHotDraw. All of the subject systems have a long development history. They are of different size and domain. Table 6.1 describes the subject systems.

- *Ant* is a Java library and command-line tool that drives processes described in build files as targets and extension points dependent upon each other. The main use of Ant is to build Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. It has over 128208 LOC.

- *dnsjava* is an open source software system, an implementation of DNS in Java. It supports all record types, and unknown record types. It can be used for queries, zone transfers, and dynamic updates. It includes a cache which can be used by clients, and a minimal implementation of a server. It supports TSIG authenticated messages, partial DNSSEC verification, and EDNS0. It has over 23738 LOC.

- *JHotDraw* is an open source software system written in Java. It is a GUI framework for technical and structured Graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. It has over 139870 LOC.

6.2.2 Data Extraction

In this section, we briefly discuss how the framework extract data from the subject systems. It automatically mines SVN repositories to check commit messages of each subject system and identify fault fixing revisions using prior fault studies [111], [46], since we do not need all the revisions of a subject system. Then, it

finds out intermediate revisions. For example, if revision r is a fault fixing revision, then the immediate revision $r-1$ is the intermediate revision. The reason to choose the intermediate revision is to approximate buggy clone classes, because we are not interested in finding out the origin of a buggy code fragment. The framework uses the *Diff* algorithm to identify changes made to fix bugs, and if any clone class is changed to fix bugs, we called that clone class a buggy clone class. It detects clones from fault fixing revisions and intermediate revisions. Details about the technique can be found in Chapter 3. We discuss clone detection process below.

6.2.3 Clone Detection

Source Code Preparation

We are not interested in those files that may come up with false positive clones. We remove all uninteresting (e.g., test files) files from the subject systems. To identify test files we checked their file names, function names and comments. They could create false positive clones, because they changed frequently for testing purpose.

Clone Detection Tool

As we discussed in Section 3.3.1, to detect clones from all the revisions, we gave preferences to NiCad [22] because it has already been shown to be effective for detecting near-miss clones while maintaining high precision and recall [103], [104], [101]. We carefully chose parameters for NiCad as described in Table 3.1. We set granularity to functions, minimum clone length to 5 LOC to avoid getter/setter methods, dissimilarity threshold to 30% and we also applied consistent renaming. Before detecting clones we process all releases of the subject systems to remove all test files so that we do not get false positive clones in our experiment.

Mapping Clone Classes

We use the framework to map clone classes between fault fixing revisions and intermediate revisions. First, we process XML outputs generated by NiCad. NiCad gives clone classes along with their file name and line numbers. Since we set granularity of NiCad to functions, it gives us all functions of fault fixing revisions and intermediate revisions. To map clones between an intermediate revision and a fault fixing revision, first we map functions using their name, signature and directory information. We use Longest Common Subsequence Count (LCSC) to find the origin of a function. Once we have a mapping for functions, we can map clone classes, this is because each clone fragment of a clone class is a function. Assume, we have a fault fixing revision r and an intermediate revision $r-1$ and pre processed clone classes of revision r and $r-1$. Then to map clone classes, first, we map all functions generated by NiCad from revision $r-1$ to revision r. As we have mapping for functions between revision $r-1$ and revision r, for each clone class of a revision $r-1$, we check the clone fragments of each clone class of r until we find the match. Details of our mapping technique can be found in the Section 3.3.3.

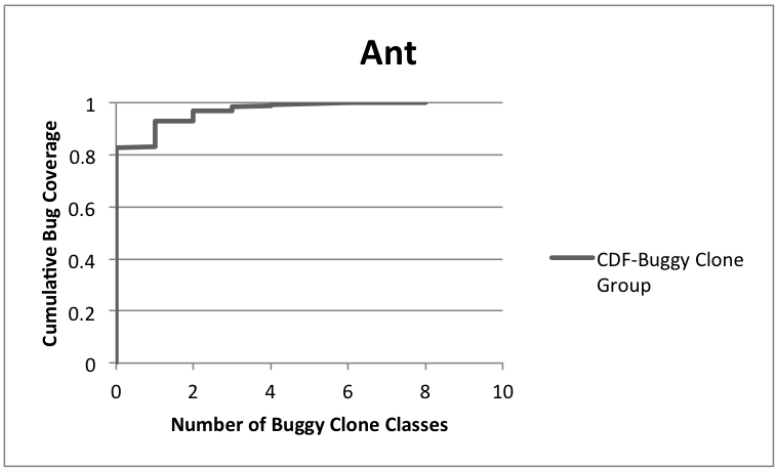


Figure 6.1: Cumulative distribution of buggy clone classes in Ant

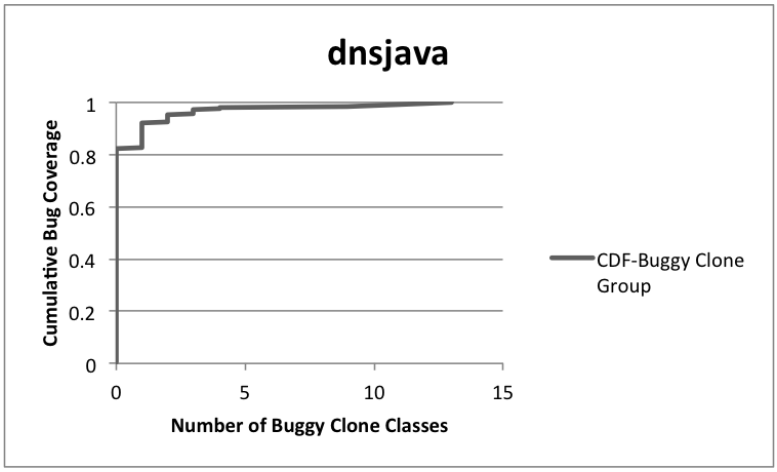


Figure 6.2: Cumulative distribution of buggy clone classes in dnsjava

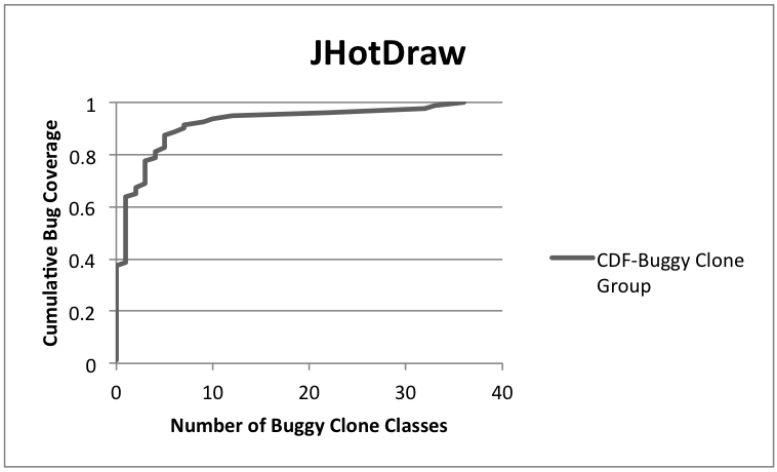


Figure 6.3: Cumulative distribution of buggy clone classes in JHotDraw

6.3 Case Study and Results

In this section we briefly discuss our results with the following research questions.

6.3.1 RQ1: To what extent are buggy clone classes related to bugs?

In this study, we are interested to know if most of the clone classes are related to a bug. The framework automatically identifies all the buggy clone classes and the non-buggy clone classes from the intermediate revisions and the fault fixing revisions of the subject systems. Then, for each subject system we plot cumulative distribution of buggy clone classes on a graph for further analysis. Figure 6.1, 6.2, and 6.3 represent the cumulative distribution of buggy clone classes.

From Figure 6.1, we see that there is about 80% chance that no clone class will contribute to a bug in Ant. We also notice that dnsjava has similar patterns to Ant. However, we can see from Figure 6.3 that in JHotDraw, there is only a 40% chance that no clone will contribute to a bug. Alternatively, we can see that most of the time there is some clone classes that are responsible for bugs. We also investigate how many buggy clone classes are there in all the releases of the subject systems. We find that there are only 10 buggy clone classes in all the releases of Ant and 15 buggy clone classes in all the releases of dnsjava. But, we find 40 buggy clone classes in all the releases of JHotDraw. Finally, we conclude that clones are not really a 'bad smell' but our findings also do not support the claim of Rahman et al. [98] that we can clone, and breathe easy, at the same time.

6.3.2 RQ2: How are buggy clones managed?

We automatically identify change patterns of clone classes from a intermediate revision to a fault fixing revision as we described in Section 3.3.3. After identifying change patterns of mapped clone classes, we checked buggy clone classes to see whether they changed consistently or inconsistently to fix a bug because if a programmer forgets to change one or more clone fragments, it may reproduce the same bug again. If a clone class changed consistently then we can say that the same changes propagated to all clone fragments and bugs are fixed. We also manually investigated randomly chosen buggy clone classes in all subject systems using our prototype.

We see from Table 6.2 that in all subject systems except dnsjava most of the clone classes were changed inconsistently. In both Ant and JHotDraw, more than 55% of buggy clone classes were changed inconsistently. In JHotDraw, more than 67% of buggy clone classes were changed inconsistently. Furthermore, we have seen that, in both Ant and dnsjava, more than 14% of buggy clone classes disappeared inconsistently because of extensive inconsistent changes. So, we can say that in Ant, 86.37% of all buggy clone classes were changed inconsistently and in JHotDraw, 70.06% of all buggy clone classes were changed inconsistently. However, in dnsjava we find a different scenario. Most of the buggy clone classes in dnsjava were changed consistently. Only, 13.21% of all buggy clone classes were changed inconsistently. We can see a dramatic fall in the number of buggy clone classes and the non-buggy clone classes in the Figure 6.5. Perhaps, a lot of refactoring was done.

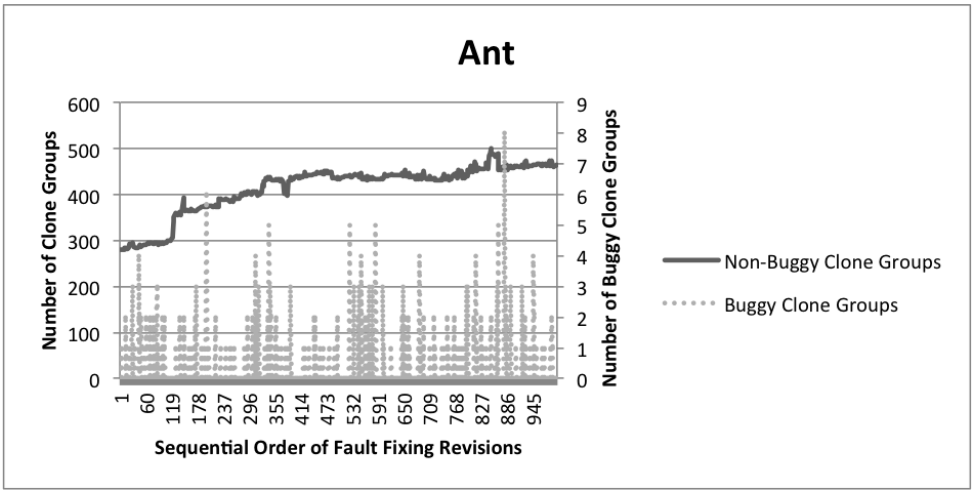


Figure 6.4: Non-Buggy clone classes vs. buggy clone classes in Ant

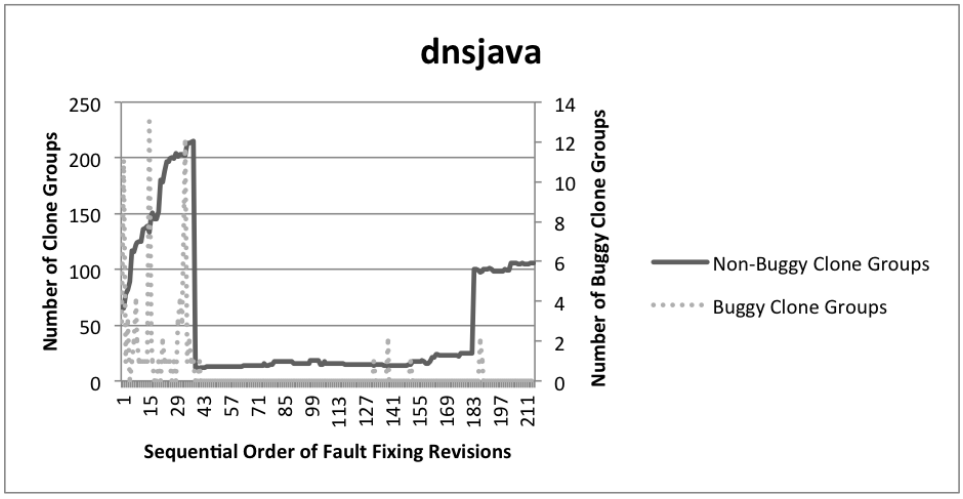


Figure 6.5: Non-Buggy clone classes vs. buggy clone classes in dnsjava

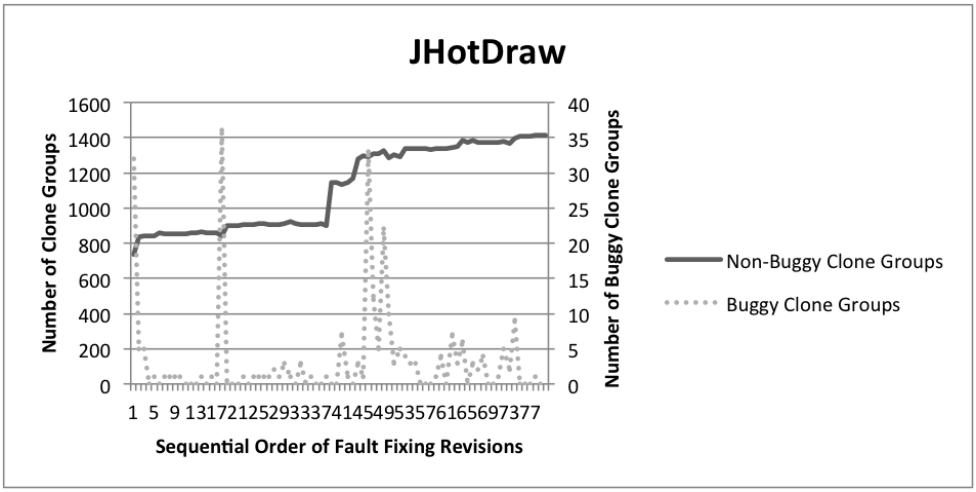


Figure 6.6: Non-Buggy clone classes vs. buggy clone classes in JHotDraw

Table 6.2: Change Patterns of Buggy Clones (Consistent Change = CC, Inconsistent Changes = IC, Disappeared Inconsistently = DI)

Subject	# of CC	% of CC	# of IC	% of IC	# of DI	% of DI	Total % of IC
Ant	15	13.63%	74	67.27%	21	19.10%	86.37%
dnsjava	92	86.79%	8	7.55%	6	5.66%	13.21%
JHotDraw	130	44.52%	119	55.33%	43	14.73%	70.06%

Table 6.3: Type Changes due to Bug Fix

Change	Ant	dnsjava	JHotDraw
Type-1− >Type-2	1	1	1
Type-1− >Type-3	3	0	13
Type-2− >Type-1	0	1	2
Type-2− >Type-3	3	0	1
Type-3− >Type-1	3	2	6
Type-3− >Type-2	1	1	4
No Type Change	106	104	285

We investigated that if any buggy clone classes change their clone types due to a bug fix. Table 6.3 shows our results. We find that some clone classes were changed due to a bug fix. We manually investigated these clone classes to see why they changed their clone types. In most cases, developers forget to change some clone fragments. To decide whether a programmer did a mistake or not, we consider the context of the code fragments. Even if dnsjava managed clones well our finding shows that developers did not manage the buggy clone classes properly in the other two systems.

We also manually investigated randomly chosen buggy clone classes using the prototype. In Figure 6.7, we show an example of a buggy clone class that was changed consistently to fix a bug. It was fixed in revision 763 and the commit message is "Activity view should not show note when there is a warning or an error". The left clone class in Figure 6.7 is taken from jhotdraw7/src/main/java/org/jhotdraw/gui/JActivityView.java of intermediate revision (revision 762). First clone fragment was started at line 155 and ended at 163 and second clone fragment was started at 145 and ended at 153. The right clone fragment showing the changes made to fix the bug. The clone class was showing 'Activity View' wrongly, when there was a warning or error. Then they were modified to consistently to stop showing 'Activity View', when there is a warning or error.

As we found most of the buggy clone classes were changed inconsistently, we were very interested to see that if these inconsistent changes were made unintentionally. We randomly choose samples from all subject systems to manually investigate them using our prototype. We found that most inconsistent changes are done unintentionally. Figure 6.8 shows *diff* of all clone fragments of a buggy clone class of JHotDraw that changed

File: jhotdraw-762/jhotdraw7/src/main/java/org/jhotdraw/gui/JActivityView.java
Revision: 762

```
private void updateError() {
    String txt = model.getError();
    errorLabel.setText(txt);
    boolean b = txt != null;
    if (errorLabel.isVisible() != b) {
        errorLabel.setVisible(b);
        revalidate();
    }
}


private void updateWarning() {
    String txt = model.getWarning();
    warningLabel.setText(txt);
    boolean b = txt != null;
    if (warningLabel.isVisible() != b) {
        warningLabel.setVisible(b);
        revalidate();
    }
}
```

File: jhotdraw-763/jhotdraw7/src/main/java/org/jhotdraw/gui/JActivityView.java
Revision: 763

```
private void updateError() {
    String txt = model.getError();
    errorLabel.setText(txt);
    updateLabelVisibility();
}


private void updateWarning() {
    String txt = model.getWarning();
    warningLabel.setText(txt);
    updateLabelVisibility();
}

private void updateLabelVisibility () {
    boolean isError = model.getError () != null;
    boolean isWarning = model.getWarning () != null;
    errorLabel.setVisible (isError);
    warningLabel.setVisible (! isError && isWarning);
    noteLabel.setVisible (! isError && ! isWarning);
    revalidate ();
}
```

Figure 6.7: Example of a Buggy Clone class in JHotDraw that was consistently changed to fix a fault

inconsistently to fix a fault. The revision 597 was committed to fix two faults and a part of the commit message was "Fixed menu creation when a JMenu has no text". The buggy clone classes contain three clone fragments in three different files. The file names are jhotdraw7/src/main/java/org/jhotdraw/app/MDIApplication.java, jhotdraw7/src/main/java/org/jhotdraw/app/OSXApplication.java, and jhotdraw7/src/main/java/org/jhotdraw/app/SDIApplication.java and start lines and end lines of these clones are 384 and 453, 300 and 369, 249 and 318. From Figure 6.8, we see that to fix a menu creation when `Jmenu` has no text first two clone fragments were changed consistently. The text value of the `mm` variable was stored in `text` to check for a `null` value. If `text` is `null` then the text value of `mm` was set to a value. Unfortunately, a programmer forgot to set the text value of variable `mm` in the last clone fragment. It created another bug. We also found that most of the inconsistent changes were done to these clone classes, which were in separate files. Finally, we can say that even if the number of buggy clone classes is very low, they are not managed well. It could be a quality issue in a software system.

6.3.3 RQ3: Is there any relationship between the growth of buggy clone classes and the growth of non-buggy clone classes over time?

In order to investigate this question, we count the number of non-buggy and buggy clone classes for the revisions, and plot them on a graph. Then, we analyzed their patterns if the number of buggy clone classes increase with the number of non-buggy clone classes during the evolution of the software systems. We also find Pearson's correlation co-efficient ($\rho_{(X,Y)}$) and p-value of each subject system. Pearson's correlation coefficient measures the strength and direction of linear relation between two variables (e.g: the number of non-buggy clone classes and the number of buggy clone classes). It give values between -1 to 1. The value of 1 for $\rho_{(X,Y)}$ indicates that there is strong positive relationship between two variables if the value of $\rho_{(X,Y)}$

Table 6.4: Statistical Analysis of the Non-buggy Clone classes and the Buggy Clone classes

Subject	$\rho_{(X,Y)}$	$p - value$
Ant	0.207	1.9024e-11
dnsjava	-0.061	0.1
JHotDraw	-0.21	0.02

```
< if (mm.getText().equals(fileMenuText)) {
---
> String text = mm.getText();
> if (text == null) {
>     mm.setText("-null-");
> } else if (text.equals(fileMenuText)) {
```

```
< if (mm.getText().equals(fileMenuText)) {
---
> String text = mm.getText();
> if (text == null) {
>     mm.setText("-null-");
> } else if (text.equals(fileMenuText)) {
```

```
< if (mm.getText().equals(fileMenuText)) {
---
> String text = mm.getText();
> if (text == null) {
> } else if (text.equals(fileMenuText)) {
```

Figure 6.8: Example of a Buggy Clone class in JHotDraw that was changed inconsistently to fix a fault

is -1 that indicates there is strong negative relationship between two variables. The $p - value$ is a statistical value that details how much evidence there is to reject the most common explanation for the data set.

We investigate how the buggy clone classes and the non-buggy clone classes over time since we know that clones increase over time [81]. Figure 6.4, 6.5 and 6.6 show result of Ant, dnsjava and JHotDraw respectively. From the results, we can see that during the evolution of all the subject systems, the number of non-buggy clone classes increased over time, but the number of buggy clone classes did not increase over time. However, we can also see some random peaks over time, which indicate that over time developers created some new buggy clone classes; however, they fixed the bugs later.

We perform statistical analysis for each subject system. Table 6.4 represent statistical data. We can see that Pearson's correlation coefficients are not strong. In JHotDraw, the value of $\rho_{(X,Y)}$ ranges between -0.3 to -0.1, which means there is small negative relationship between the non-buggy clone classes and the buggy clone classes. The dnsjava show no relationship between the non-buggy clone classes and the buggy clone classes. Because, the value of $\rho_{(X,Y)}$ ranges between -0.09 and 0.0. However, in the Ant, we find small positive relationship between them, which is negligible.

We also calculated $p - value$ for all subject systems to check significance of our data. In Table 6.4, right most column represents $p - value$ of all subject systems. The $p - value$s are significant for both the Ant and the JHotDraw. However, the $p - value$ for dnsjava is not significant. It's may be because extensive refactoring was done to reduce clones, which was different than the other systems. From Figure 6.5, we can see that the number of non-buggy clone classes was fell down dramatically, then again increased at the end. But, from Pearson's correlation coefficient value we can say there is no relationship between these two clone classes.

Finally, we can conclude that there is no strong relationship between the cloning rate of the buggy clone classes and the non-buggy clone classes over time.

6.3.4 RQ4: Which category of buggy clone classes are more buggy from others?

Table 6.5: Categories of Buggy Clone classes in Terms of the Numbers of Clone Fragments. BCG = Buggy Clone classes

Subjects	Small			Medium			Large		
	# of BCG	Total Clone classes	%	# of BCG	Total Clone classes	%	# of BCG	Total Clone classes	%
Ant	101	378256	0.02%	8	20755	0.04%	8	13831	0.06%
dnsjava	98	11090	0.88%	11	441	2.49%	0	141	0%
JHotDraw	288	84790	0.34%	18	3089	0.58%	6	1135	0.52%

First, we identified the buggy clone classes. Then we categorize them based on the number of clone fragments. The 'Small' clone classes contain 2-5 clone fragments, the 'Medium' clone classes contain 6-10 clone fragments, and the 'Large' clone classes contain more than 10 clone fragments. Then we investigate which categories of clone classes are more buggy than the other categories. We categorize clone classes in terms of the number of clone fragments to see which category of clone classes are more buggy than others. Table 6.5 shows our result. We see from the table that the number of 'Small' buggy clone classes were high, but the percentage of medium buggy clone classes were higher than others in all subject systems. There were a few 'Large' buggy clone classes. So, we can say that 'Small' and 'Medium' clone classes are more buggy than others.

6.4 Contribution of the Framework and Prototype

Since our framework can automatically mine software repositories and find fault fixing revisions as well as genealogies of a subject system, we take full advantage of our framework to conduct this study to know how clones contribute to bug. We use the JSON output of the framework to analyze how clones can contribute to bugs. We extensively use the prototype to manually investigate how buggy clone classes were changed to fix bugs. We also investigate how many clone fragments a buggy clone class contains in general using the filtering options of our prototype easily.

6.5 Threats To Validity

We collected all the bugs of Ant from bugzilla. We collected all bug reports from http://sourceforge.net/ for dnsjava and JHotDraw. They may not represent a complete reference of all bugs. On the other hand, to find out a fault fixing revision, we ran our program. After extensive manual investigation, we made sure that the accuracy was above 89%. Thus, we may miss some fault fixing revisions.

In our study we may have some false positive buggy clone classes, because we detect buggy clone classes based on the *diff* algorithm. We check the changes made to fix faults and then we track the line numbers to

check if any clone class was changed. It may happen that a developer modified some lines before a commit that are clones and they are not related to bugs, then we may have some false positive buggy clone classes. However, it would not have a negative impact, because if there exists some false positive buggy clone classes, then we can say that the number of buggy clone classes are less than what we got but not more than that.

Our study directly depends on raw clone data, which depends on a clone detection tool and the parameters we used to detect clones. In order to mitigate this threat, we carefully chose the NiCad clone detector which has been found to be an effective clone detector to detect exact and near-miss clones [103], [104], [101]. The settings we used to detect clones has shown very high high precision and recall in previous studies [101], [104]

We automatically map clones from the intermediate revision to the fault fixing revision. In that case, some clones can be mapped wrongly, but our manual investigation indicates that this is very rare.

In our case study we choose only three open source software systems from different application domains. Our result may not be generalizable to other software systems. Probably, by adding more subject systems this problem can be solved. We plan to add more subject systems in a future study.

6.6 Summary

Programmers often clone source code intentionally or unintentionally. Therefore, we need to know whether cloning is good or bad. We investigate three open source software systems of different domains and size utilizing our framework and prototype to find whether code cloning is bad or not. We represent our results by answering four research questions.

The subject systems we used for our study were of medium and large size. We find that there is as little as 40% chance that there will be no buggy clone classes in a subject system. Therefore, we can say even if clones are not as bad as we think but still we should be careful because most of the time there is some clone classes that are responsible for bugs.

We look in depth into buggy clone classes to see how they change to fix bugs. Because, inconsistent changes of clone classes may lead to bugs. We find that more than 70% of buggy clone classes changed inconsistently. In both Ant and JHotDraw most of the buggy clone classes were changed inconsistently. In Ant, 86.37% of all buggy clone classes were changed inconsistently. However, in dnsjava more than 86% buggy clone classes were changed consistently, it may be because it was refactored extensively. Furthermore, from our manual investigation we can say that inconsistent changes in buggy clone classes may reproduce the same bug or lead to other bugs. We also find that most of the buggy clone classes that are in separate files were changed inconsistently and some buggy clone classes changed their type due to fixed bugs. In most cases, the changes were unintentional. Thus, we cannot say that programmers are capable of remembering all the buggy clones as we find that most of the time they change buggy clone classes inconsistently.

We performed statistical analysis on the data to see if there is any strong relation between the growth the non-buggy clone classes and buggy clone classes. We determined Pearson's correlation coefficient and

$p - value$ from the data. We find no strong relationship between the growth of non-buggy clone classes and buggy clone classes. So, we can say that the number of buggy clone classes do not increase over time with the number of non-buggy clone classes.

We also categorize buggy clone classes in terms of the number of clone fragments. We find that the 'Small' and the 'Medium' categories of buggy clone classes are more buggy than that the 'Large' category. Alternatively, we say that most of the buggy clone classes contain 2 to 10 clone fragments.

Finally, we conclude that we can clone to speed up the development process but we need to manage buggy clone classes properly. We also show that our framework and prototype are useful for managing buggy clone classes. The prototype is especially useful for investigating how buggy clone classes change over time. In the future, we are planning to study more subject systems covering different languages and domains, so that we can generalize our findings to other open source software systems.

Chapter 7

Conclusion

Managing clones is an inevitable part of software maintenance as long as it is practically impossible to remove all of the clones from a software system [70]. However, it would be difficult to manage clones without understanding how code clones evolve in a software system over time. Therefore, a clone management system is required that can help us to understand the clone genealogies in software systems and to find patterns so that we can find and manage problematic clones easily.

7.1 Thesis Statement

In this research, we propose a framework for extracting and visualizing clone genealogies in a software system, which we use to build a prototype for a multi-touch surface and use to elicit feedback from practicing researchers and developers. Both the framework and the prototype help us to efficiently find clone patterns reducing the investment in time and effort, which in turn helps us to manage clones. To validate the usefulness of the framework and the prototype, we conduct two empirical studies and represent our findings by answering a number of research questions that requires a detailed investigation of the clones supported by the prototype.

7.2 Contributions and Results

In this research, we propose a framework for extracting and visualizing clone genealogies in software systems. Since most of the studies of code clone evolution are limited to the Type-1 and Type-2 clones, we consider Type-1, Type-2 as well as Type-3 clones for constructing clone genealogies. The framework uses several techniques to provide information of a clone genealogy and the information includes change patterns, genealogy type, bug information, developer information, information of each instances of a clone class in a genealogy, etc. Furthermore, we provide a visualization model with a detailed data organization so that it can be used for developing a visualization tool. The framework produces JSON output so that it can be easily parsed for further analysis. We also compare our framework with the gCad framework [109] and the comparison shows that our framework is able provide significantly more information than gCad. The more information we get, the more patterns we will be able to identify and the more patterns we identify, the better we will be able to manage clones in software systems. Therefore, the framework would help us to understand and manage clones more efficiently.

Since our framework can be used for developing a visualization tool for visualizing clone genealogies, we incorporated our framework into a prototype for a multi-touch surface to visualize clone genealogies in software systems. We use interactive user interfaces for visualizing clone genealogies that will help us to better understand how clone classes evolve in a software system with less time and less effort. We chose colors of each interfaces carefully so that all the colors are perceivable by people with normal vision and people with common color vision deficiencies (CVDs). The prototype allows a user to find genealogies based on some specific criteria. The ability of visualizing source code changes across revisions and between clone fragments in a clone class makes the prototype more useful for manual investigation. A user also can annotate a clone fragment for future analysis and can contact developers if necessary. We conducted structured and semi-structured interviews to elicit feedback from researchers and developers. And, the outcomes of the interviews show that in most cases, they found the prototype useful. We present their feedback and comments for future improvement of the prototype.

We extend our framework to conduct an empirical investigation into the evolution of function clones using three Java open source software systems. We classified function clones into five categories base on function types. Our findings show that developers have the tendency of creating FCType-2 clones as well as FCType-4 clones. The number of FCType-2 clone classes gradually increases across releases and it grows faster than the other categories of clone classes. We also find that there is about 53% to 93% of long lived FCType-2 genealogies and 51% to 82% of FCType-4 long lived genealogies in the subject systems. Since there are FCType-2 clone genealogies ranging from 29% to as high as 39%, and FCType-4 clone genealogies ranging from 16% to 38%, and there are more FCType-2 and FCType-4 inconsistently changed clone genealogies than the FCType-2 and the FCType-4 consistently changed genealogies, we can say that they need extra attention while managing clones. We investigate how long lived clone genealogies changed during their evolution and we find that most of the time they did not change consistently. Therefore, we can try to refactor most of the long lived clone classes. We also show that about 60% to 75% of changed clone genealogies were converted to FCType-2, which could be a reason for a large amount of FCType-2 clones across releases. This study also validate our claim that the framework and prototype are useful for finding patterns easily and efficiently.

We conducted a second empirical study to find a relationship between clones and bugs using three Java software systems with the help of our prototype. We find that there is as little as 40% chance that there will be no buggy clone classes in a subject system. Therefore, we should be careful when creating new clones. We use the prototype to identify change patterns of the buggy clone classes and for an extensive manual investigation. We show that most of the buggy clone classes changed inconsistently and developers often forget to propagate changes to clone fragments in different files. We investigate whether the number of buggy clone classes increases with the number of non-buggy clone classes and our statistical analysis shows that there is no strong relationship between the growth of the buggy clone classes and the non-buggy clone classes. Our findings also show that generally buggy clone classes with 2 - 10 clone fragments are more prone to bugs. By conducting this study, we also show that our framework and prototype are useful for managing buggy

clone classes. The prototype is especially useful for investigating how buggy clone classes change over time.

7.3 Future Work

In this section, we provide directions towards future research and further improvement of the prototype that can help us to manage clones more efficiently.

7.3.1 Improvement of the Prototype

Our prototype is useful for finding patterns, investigating clone genealogies, contacting developers, etc. However, there is still room for improvement. In the future, multi developer feature can be added so that multiple developers can analyze clone genealogies on different surfaces at a time. This would be useful for making refactoring decisions faster for the projects that involve multiple developers and for reducing unintentional clones that are often created while implementing the same functionality. Visualization of *Diff* can be improved. The prototype only allows a user to see an inline Diff of two clone fragments. Side-by-side Diff can be implemented so that a user can use whichever they prefer. A prototype could be built for larger surfaces (e.g., table Tops) using the framework which may be suitable for teams to analyze clone genealogies more efficiently.

7.3.2 IDE Based Visualization

Since the framework incorporates a visualization model with a detailed data organization, it could be used to implement a plugin for IDEs. We also presented user interface ideas in Chapter 4 and these could be used for designing user interfaces for IDEs. The available tool support in IDEs is still inadequate.

7.3.3 API Analysis in Clone Genealogies

We need to find more patterns to better manage clones in software systems. We could investigate which APIs are mostly responsible for creating clones in a subject system. We can conduct this study based on programming languages to distinguish those APIs based on programming languages. Then, we can see their genealogies and how they change over time. For example, if we see that the APIs dealing with files are responsible for creating clones then we can think of refactoring them.

7.3.4 Clones and Bugs

In this research, we conducted the second empirical study to see how clones contribute to bugs but it would be interesting to conduct the study on a larger scale to see to what extent cloning is safe, since sometimes a single bug can be a critical issue.

References

[1] Scalable and Incremental Clone Detection for Evolving Software. *ICSM*, 2009.

[2] Eytan Adar. Guess: a language and interface for graph exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 791–800, New York, NY, USA, 2006. ACM.

[3] Eytan Adar and Miryung Kim. Softguess: Visualization and exploration of code clones in context. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 762–766, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *Across Software Systems.International Symposium on Empirical Software Engineering (ISESE'05*, pages 376–385, 2005.

[5] Ghazi Alkhatib. The maintenance problem of application software: an empirical analysis. *Journal of Software Maintenance*, 4(2):83–104, June 1992.

[6] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 273–280. PDF, 2001.

[7] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44:755–765, 2002.

[8] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[9] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95, pages 86–, Washington, DC, USA, 1995. IEEE Computer Society.

[10] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimóthy. Clone smells in software evolution. In *23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France*, pages 24–33. IEEE, 2007.

[11] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 326–, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 98–, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Liliane Barbour, Foutse Khomh, and Ying Zou. Late propagation in software clones. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 273–282, Washington, DC, USA, 2011. IEEE Computer Society.

[14] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 85–94, Washington, DC, USA, 2009. IEEE Computer Society.

[16] J. Birch. *Diagnosis of defective colour vision*. Butterworth-Heinemann Limited, 2001.

[17] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: A semiautomated approach, 2004.

[18] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[19] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 7–13, New York, NY, USA, 2011. ACM.

[20] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines for text and code. *Journal of American Statistical Association, Institute for Mathematical Statistics and Interface Foundations of North America*, 2(2):153–174, June 1993.

[21] James R. Cordy. Comprehending reality–practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–, Washington, DC, USA, 2003. IEEE Computer Society.

[22] James R. Cordy and Chanchal K. Roy. The nicad clone detector. In *ICPC*, pages 219–220, 2011.

[23] James R Cordy and Chanchal K Roy. Tuning research tools for scalability and performance: The nicad experience. *Science of Computer Programming*, page 26, 2013 (to appear).

[24] Neil Davey, Paul Barson, Simon Field, Ray Frank, and Stewart Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, (1):219–236, 1995.

[25] Ian J. Davis and Michael W. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 77–78, New York, NY, USA, 2010. ACM.

[26] Ian J. Davis and Michael W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *WCRE*, pages 242–246, 2010.

[27] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Michael Pfaehler, and Bernhard Schätz. Model clone detection in practice. In *IWSC*, pages 57–64, 2010.

[28] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-Francois Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *ICSE*, pages 603–612, 2008.

[29] Christoph Domann, Elmar Jürgens, and Jonathan Streit. The curse of copy&paste cloning in requirements specifications. In *ESEM*, pages 443–446, 2009.

[30] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20(1):3:1–3:31, July 2010.

[32] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–, Washington, DC, USA, 1999. IEEE Computer Society.

[33] Stephen G. Eick, Paul Schuster, Audris Mockus, Todd L. Graves, and Alan F. Karr. Visualizing software changes. *INTERACTIONS*, 17:29–31, 2002.

[34] Richard Fanta and Vclav Rajlich. Removing clones from the code. *Journal on Software Maintenance and Evolution*, 11(4):223–243, July/Aug. 1999.

[35] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[36] David Flatla and Carl Gutwin. "so thats what you see!" building understanding with personalized simulations of colour vision deficiency. In *ASSETS '12: The proceedings of the 14th international ACM SIGACCESS conference on Computers and accessibility*, Boulder, Colorado, USA, 2012.

[37] M. Fowler and K. Beck. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[38] Nils Göde. Clone removal: fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 33–40, New York, NY, USA, 2010. ACM.

[39] Nils Göde and Jan Harder. Oops! . . . i changed it again. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 14–20, New York, NY, USA, 2011. ACM.

[40] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 311–320, New York, NY, USA, 2011. ACM.

[41] Nils Göde and Rainer Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.

[42] Nils Göde, Nils and Jan Harder. Clone stability. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 65–74, Washington, DC, USA, 2011. IEEE Computer Society.

[43] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 103–106, New York, NY, USA, 2001. ACM.

[44] Jan Harder and Nils Göde. Modeling clone evolution. In *3rd International Workshop on Software Clones*, pages 17–21, 2009.

[45] Jan Harder and Nils Göde. Efficiently handling clone data: Rcf and cyclone. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 81–82, New York, NY, USA, 2011. ACM.

[46] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[47] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Refactoring support based on code clone analysis. In *PROFES*, pages 220–233, 2004.

[48] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Aries: refactoring support tool for code clone. In *Proceedings of the third workshop on Software quality*, 3-WoSQ, pages 1–4, New York, NY, USA, 2005. ACM.

[49] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Code clone analysis methods for effcient software maintenance. Technical report, Graduate School of Information Science and Technology, Osaka University, 2006.

[50] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In *In Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES'02*, pages 185–197. Springer, 2002.

[51] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 73–82, New York, NY, USA, 2010. ACM.

[52] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 55–64, New York, NY, USA, 2007. ACM.

[54] Zhen Ming Jiang and Ahmed E. Hassan. A framework for studying clones in large software systems. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 203–212, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Zhen Ming Jiang, Ahmed E. Hassan, and Richard C. Holt. Visualizing clone cohesion and coupling. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, APSEC '06, pages 467–476, Washington, DC, USA, 2006. IEEE Computer Society.

[56] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[57] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, Washington, DC, USA, 1994. IEEE Computer Society.

[58] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '94, pages 32–. IBM Press, 1994.

[59] J. Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pages 16–. IBM Press, 1996.

[60] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Nicolas Juillerat and Beat Hirsbrunner. An algorithm for detecting and removing clones in java code. In *Software Evolution through Transformations*, pages 63–74, 2006.

[62] Elmar Jürgens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schätz, Stefan Wagner, Christoph Domann, and Jonathan Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE (2)*, pages 79–88, 2010.

[63] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[64] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of the Principles of Software Evolution, 7th International Workshop*, IWPSE '04, pages 85–94, Washington, DC, USA, 2004. IEEE Computer Society.

[65] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.

[66] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. *Reverse Engineering, Working Conference on*, 0:19–28, 2006.

[67] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: Research articles. *J. Softw. Maint. Evol.*, 18(2):61–82, March 2006.

[68] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, December 2008.

[69] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, ISESE '04, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[70] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE'05*, pages 187–196, 2005.

[71] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[72] Raghavan Komondoor and Susan Horwitz. Effective, automatic procedure extraction. In *IWPC*, pages 33–, 2003.

[73] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[74] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–, Washington, DC, USA, 2001. IEEE Computer Society.

[75] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Jens Krinke. Is cloned code more stable than non-cloned code? In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 28-29 September 2008, Beijing, China*, pages 57–66. IEEE, 2008.

[77] Jens Krinke. Is cloned code older than non-cloned code? In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 28–33, New York, NY, USA, 2011. ACM.

[78] Bruno Lague, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 314–, Washington, DC, USA, 1997. IEEE Computer Society.

[79] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, September 2003.

[80] António Menezes Leitão. Detection of redundant code using r2d2. *Software Quality Control*, 12(4):361–382, December 2004.

[81] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[82] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, March 2006.

[83] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, APSEC '06, pages 269–276, Washington, DC, USA, 2006. IEEE Computer Society.

[84] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pages 227–236, 2008.

[85] Angela Lozano and Michel Wermelinger. Tracking clones' imprint. In *IWSC*, pages 65–72, 2010.

[86] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 18–, Washington, DC, USA, 2007. IEEE Computer Society.

[87] Giuseppe A. Di Lucca, Massimiliano Di Penta, and Anna Rita Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 481–486, Washington, DC, USA, 2002. IEEE Computer Society.

[88] Udi Manber. Finding similar files in a large file system. In *USENIX WINTER 1994 TECHNICAL CONFERENCE*, pages 1–10, 1994.

[89] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 107–, Washington, DC, USA, 2001. IEEE Computer Society.

[90] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–, Washington, DC, USA, 1996. IEEE Computer Society.

[91] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[92] Manishankar Mondal, Chanchal K. Roy, Md. Saidur Rahman, Ripon K. Saha, Jens Krinke, and Kevin A. Schneider. Comparative stability of cloned and non-cloned code: an empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1227–1234, New York, NY, USA, 2012. ACM.

[93] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. Dispersion of changes in cloned and non-cloned code. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 29–35. IEEE, 2012.

[94] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. An empirical study on clone stability. *ACM SIGAPP Applied Computing Review*, 12(3):20–36, 2012.

[95] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 87–, Washington, DC, USA, 2002. IEEE Computer Society.

[96] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[97] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[98] Foyzur Rahman, Christian Bird, and Premkumar T. Devanbu. Clones: What is that smell? In *MSR*, pages 72–81, 2010.

[99] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *In WCRE*, pages 100–109, 2004.

[100] Chanchal K. Roy. Detection and analysis of near-miss software clones. In *ICSM*, pages 447–450, 2009.

[101] Chanchal K. Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[102] Chanchal K Roy and James R Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings of the 2008 C 3 S 2 E conference*, pages 137–140. ACM, 2008.

[103] Chanchal K. Roy and James R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *ICST Workshops*, pages 157–166, 2009.

[104] Chanchal K. Roy and James R. Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance*, 22(3):165–189, 2010.

[105] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *School of Computing Technical Report 2007-541, Queen's University*, page 115, 2007.

[106] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[107] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[108] Ripon K. Saha, Muhammad Asaduzzaman, Minhaz F. Zibran, Chanchal K. Roy, and Kevin A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.

[109] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 293–302, Washington, DC, USA, 2011. IEEE Computer Society.

[110] Ripon K. Saha, Chanchal K. Roy, Kevin A. Schneider, and Dewayne E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 139–148, Piscataway, NJ, USA, 2013. IEEE Press.

[111] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 249–258, Washington, DC, USA, 2010. IEEE Computer Society.

[112] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[113] Harald Störrle. Towards clone detection in uml domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 285–293, New York, NY, USA, 2010. ACM.

[114] Robert Tairas and Jeff Gray. Sub-clones: Considering the part rather than the whole. In *Software Engineering Research and Practice*, pages 284–290, 2010.

[115] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 50–54, New York, NY, USA, 2006. ACM.

[116] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[117] Md Sharif Uddin, Chanchal K Roy, and Kevin A Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Proceedings of the Tool Demonstration Track of the 21st IEEE International Conference on Program Comprehension (ICPC 2013)*, pages 236–238. IEEE, 2013.

[118] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 13–22, Washington, DC, USA, 2011. IEEE Computer Society.

[119] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 67–, Washington, DC, USA, 2002. IEEE Computer Society.

[120] Minhaz F. Zibran and Chanchal K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 105–114, Washington, DC, USA, 2011. IEEE Computer Society.

[121] Minhaz F. Zibran, Ripon K. Saha, Muhammad Asaduzzaman, and Chanchal K. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *Proceedings of the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '11, pages 295–304, Washington, DC, USA, 2011. IEEE Computer Society.